Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Informatyki

Praca dyplomowa magisterska

na kierunku Informatyka
w specjalności Systemy internetowe wspomagania zarządzania

Analiza i porównanie wydajności aplikacji w Javie korzystających z wątków wirtualnych i programowania reaktywnego

Michał Tarka
Numer albumu 330531

promotor
doc. dr inż. Roman Podraza

WARSZAWA 2025

Analiza i porównanie wydajności aplikacji w Javie korzystających z wątków wirtualnych i programowania reaktywnego

Streszczenie.

Praca ta ma na celu porównanie wydajności aplikacji w Javie z wykorzystaniem dwóch różnych podejść: wątków wirtualnych i programowania reaktywnego. W obecnych czasach aplikacje powinny być zoptymalizowane pod kątem jednoczesnego przetwarzania wielu zadań, zachowując przy tym jak najmniejsze zapotrzebowanie na zasoby. Kluczową rolę odgrywają tutaj operacje blokujące, które zatrzymują wykonywanie zadania na czas trwania tej operacji.

Jednym z rozwiązań tego problemu jest paradygmat programowania reaktywnego, który skupia się na systemach asynchronicznych i nieblokujących. Jednak podejście to posiada wiele wad. Alternatywą mogą być wątki wirtualne wprowadzone w najnowszych wersjach Javy, które mają na celu zapewnienie lekkiej alternatywy dla tradycyjnych wątków, umożliwiając bardziej efektywne zarządzanie zadaniami współbieżnymi.

Badanie zaczyna się od przedstawienia teoretycznych podstaw obu podejść. Omówione zostaną ich kluczowe cechy, zalety oraz zastosowania. Następnie praca opisuje eksperymentalne środowisko, w którym zaimplementowano aplikacje oraz wyniki eksperymentów porównawczych.

Praca kończy się na analizie metryki wydajności, takich jak zużycie zasobów, czas odpowiedzi oraz skalowalność, oraz wnioskach z nich wynikających. Wyniki pokażą, w jakich warunkach każde z podejść sprawdza się najlepiej.

Słowa kluczowe: Java, programowanie reaktywne, wątki wirtualne, współbieżność, wydajność

Analysis and comparison of the performance of Java applications using virtual threads and reactive programming."

Abstract.

This work aims to compare the performance of Java applications using two different approaches: virtual threads and reactive programming. In today's world, applications should be optimized to process multiple tasks simultaneously, while keeping resource requirements as low as possible. Blocking operations, which stop the execution of a task for the duration of that operation, play a key role here.

One solution to this problem is the reactive programming paradigm, which focuses on asynchronous and non-blocking systems. However, this paradigm has many drawbacks. As an alternative, virtual threads, introduced in the latest versions of Java, can provide a lightweight alternative to traditional threads, enabling more efficient management of concurrent tasks.

The study begins by presenting the theoretical foundations of both approaches. Their key features, advantages, and applications will be discussed. Next, the work describes the experimental environment in which the applications were implemented and the results of comparative experiments.

The paper concludes with an analysis of performance metrics, such as resource consumption, response time and scalability, and the conclusions drawn from them. The results will show under what conditions each approach performs best.

Keywords: Java, reactive programming, virtual threads, concurrency, benchmarking

załącznik nr 3 do zarządzenia
nr 28 /2016 Rektora PW

Politechnika Warszawska

……..
miejscowość i data

……..
imię i nazwisko studenta

……..
numer albumu

…….…….
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta

Spis treści

1. Wstęp

1.1. Problem badawczy

Wraz z rozwojem IT coraz więcej danych jest wymienianych między aplikacjami, a znaczna ich część przesyłana jest automatycznie, bez udziału użytkownika [24]. Aplikacje komunikują się między sobą, kiedy zdiagnozują taką potrzebę, nie czekając na interakcję użytkownika. Taki sposób korzystania z infrastruktury stawia nowe wyzwania przed twórcami oprogramowania. Sprawia, że wymagania dotyczące wydajności, dostępności i skalowalności rosną [9]. Z jednej strony ilość danych globalnie rośnie, a z drugiej koszt ich przetwarzania powinien być minimalizowany, zwłaszcza w środowiskach chmurowych.

Kluczowym aspektem staje się współbieżność, czyli zdolność systemu do jednoczesnej obsługi wielu żądań. Jednym z wyzwań, które napotykają aplikacje współbieżne, są operacje blokujące, które mogą znacząco wpływać na wydajność systemu. Operacje te, takie jak oczekiwanie na odpowiedź z bazy danych, odczyt plików z dysku czy komunikacja sieciowa, wstrzymują wątek wykonawczy do momentu zakończenia danej operacji. W tradycyjnym modelu synchronicznym prowadzi to do sytuacji, w której zasoby obliczeniowe są niewykorzystane, ponieważ wątek oczekuje, zamiast wykonywać inne zadania. Pojemność puli wątków ogranicza liczbę żądań, które mogą być obsługiwane jednocześnie, co w systemach o dużym obciążeniu może ograniczać skalowalność i efektywność aplikacji [22].

Jednym z rozwiązań tego problemu jest programowanie reaktywne, które w ostatnich latach zyskiwało na popularności. Mimo że koncepcja reaktywności pojawiła się już w latach 60. XX wieku, dopiero niedawno znalazła szerokie zastosowanie w inżynierii oprogramowania. Programowanie reaktywne to paradygmat oparty na asynchronicznych strumieniach danych, umożliwiający systemom natychmiastowe reagowanie na zdarzenia w czasie rzeczywistym. Ten paradygmat znacząco różni się od tradycyjnego programowania imperatywnego, koncentrując się na przepływie danych i propagacji zmian zamiast na sekwencyjnych instrukcjach. Ma większą złożoność od rozwiązań klasycznych, a aplikacje są bardziej skomplikowane do zaimplementowania. Może to stanowić wyzwanie pod względem czytelności i zrozumienia kodu, co w efekcie może wydłużyć czas potrzebny na jego opracowanie. Zwykle programowanie reaktywne wykorzystuje się w przypadku bardziej złożonych rozwiązań oraz projektów [13], [18].

Alternatywą dla programowania reaktywnego są wirtualne wątki wprowadzone w Javie 19 w ramach projektu Project Loom. Project Loom ma na celu połączenie korzyści płynących z wydajności programowania asynchronicznego z prostotą bezpośredniego, „synchronicznego" stylu programowania. Wirtualne wątki oferują bardziej intuicyjne podejście do współbieżności, zachowując przy tym wysoką wydajność. Wirtualne wątki pozwalają na tworzenie ogromnej liczby lekkich wątków, które są efektywnie zarządzane przez maszynę wirtualną Javy i mapowane na dostępne wątki systemowe. Umożliwia to

osiągnięcie wyższej skalowalności i wydajności przy mniejszym zużyciu zasobów oraz poprawę współbieżności [21]. Eliminują problem blokowania fizycznych wątków, co prowadzi do lepszego zarządzania zasobami i wyższej wydajności systemu.

1.2. Cel pracy

Dzięki wprowadzeniu wirtualnych wątków, tradycyjne frameworki, takie jak Spring Boot, mogą efektywnie działać w aplikacjach wymagających wysokiej współbieżności bez konieczności stosowania paradygmatu reaktywnego. Programiści mogą korzystać z imperatywnego stylu programowania, który jest bardziej intuicyjny i łatwiejszy do zrozumienia. Celem pracy jest porównanie obu tych podejść.

1.3. Zakres badania

Przedmiotem badań jest porównanie wydajności, stabilności aplikacji imperatywnej oraz reaktywnej w takich samych środowiskach oraz przy użyciu tych samych scenariuszy testowych. W ramach pracy zostaną opracowane dwie aplikacje w języku Java wykorzystujące najbardziej popularny framework Spring:

- Aplikacja oparta na programowaniu reaktywnym z wykorzystaniem frameworka *Spring WebFlux*.
- Aplikacja zbudowana przy użyciu tradycyjnego modelu *Spring MVC*.

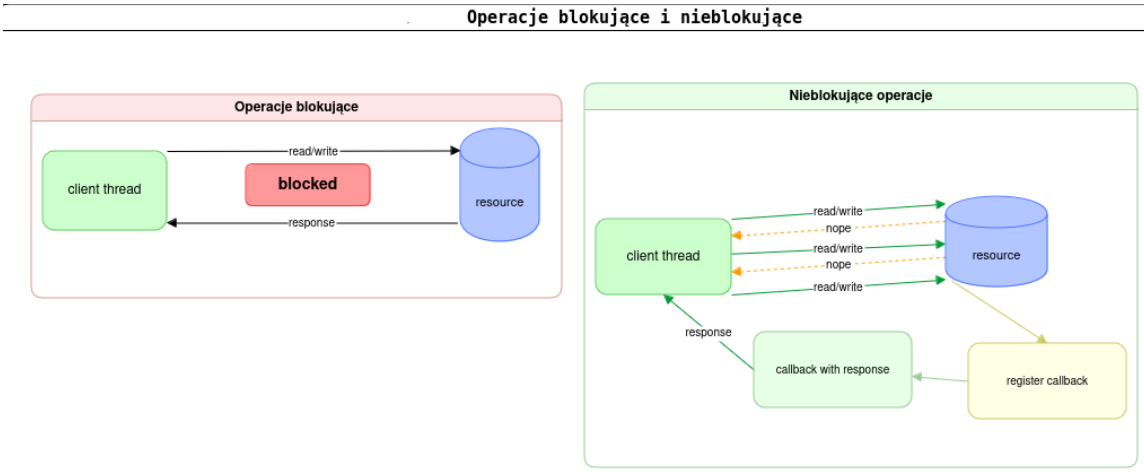
Obie aplikacje będą realizować identyczną logikę biznesową, co umożliwi bezpośrednie porównanie ich wydajności oraz zachowania w różnych warunkach obciążenia. Planowane jest przeprowadzenie serii testów wydajnościowych, które obejmą różne scenariusze. Do przeprowadzenia testów zostaną wykorzystane narzędzia umożliwiające monitorowanie parametrów i wydajności aplikacji. Zebrane dane zostaną poddane analizie w celu oceny zalet i wad wykorzystania wątków wirtualnych oraz programowania reaktywnego w tworzeniu współbieżnych aplikacji webowych w Javie. Wyniki mogą posłużyć jako praktyczne wskazówki przy wyborze odpowiedniej technologii w zależności od specyficznych potrzeb projektu.

2. Podstawy teoretyczne

W tym rozdziale opisane zostaną podstawowe definicje wymagane do zrozumienia celów projektu.

2.1. Operacje blokujące

Operacje blokujące to operacje, które wymagają udziału procesora przez cały czas ich wykonywania, nie pozwalając mu na realizację innych zadań w trakcie trwania tych operacji. Przykłady obejmują odpytywanie zewnętrznych API, gdzie wątek oczekuje na odpowiedź HTTP, oraz zapytania do baz danych, blokujące wątek aż do momentu zwrócenia wyników. Podobnie dzieje się podczas operacji plikowych, wywołań RPC czy przetwarzania wiadomości w systemach kolejkowych, jak Kafka lub RabbitMQ. Inne przypadki to operacje sieciowe, jak otwieranie połączeń TCP oraz integracje z systemami zewnętrznymi. Blokowanie wątków uniemożliwia efektywne wykorzystanie zasobów systemu, powodując ograniczenie skalowalności i obniżenie wydajności aplikacji.



Rysunek 2.1. Operacje blokujące i nieblokujące

2.2. Współbieżność

Współbieżność to zdolność do wykonywania wielu zadań jednocześnie. Oznacza to uruchamianie wielu wątków, które mogą współdzielić zasoby procesora. W praktyce działa to poprzez odpowiednie zarządzanie wieloma strumieniami wykonywania kodu, które mogą działać równolegle lub współdzielić zasoby w sposób synchroniczny. Współbieżność nie oznacza zawsze równoległości, gdyż w systemach jednordzeniowych zadania są przełączane przez planistę zadań CPU (ang. scheduler) , co daje wrażenie równoczesności. *Scheduler* kluczowy element systemu operacyjnego, zarządza kolejnością i czasem wykonywania zadań na procesorze. Stosuje algorytmy, takie jak round-robin czy priorytetowe, aby zbalansować obciążenie i zapewnić sprawiedliwy

dostęp do zasobów. Pozwala to na bardziej efektywne wykorzystanie procesora i poprawę wydajności aplikacji, szczególnie w środowiskach wielordzeniowych. Współbieżność w kontekście aplikacji internetowych może odnosić się do zdolności aplikacji do obsługi wielu żądań użytkowników w tym samym czasie [11].

2.2.1. Prawo Little'a o skalowalności

Skalowalność aplikacji serwerowych jest regulowana przez prawo Little'a [31], które odnosi się do zależności między czasem przetwarzania (latencją), współbieżnością (liczbą równocześnie przetwarzanych żądań) oraz przepustowością (liczbą obsługiwanych żądań na jednostkę czasu). Zależność tę opisuje prawo Little'a, zgodnie z którym:

$$L = \lambda \cdot W,$$

gdzie:

- L – średni poziom współbieżności (liczba aktualnie obsługiwanych żądań),
- λ – przepustowość (średnia liczba żądań na sekundę),
- W – średni czas obsługi żądania.

Przekształcając powyższe równanie, otrzymujemy:

$$\lambda = \frac{L}{W}.$$

Oznacza to, że aby zwiększyć przepustowość, należy zwiększyć liczbę obsługiwanych równolegle żądań lub skrócić czas ich obsługi. W modelu thread-per-request wiąże się to bezpośrednio z koniecznością posiadania większej liczby wątków. Jeżeli średni czas obsługi żądania jest stały, to w celu dziesięciokrotnego zwiększenia przepustowości należałoby dziesięciokrotnie zwiększyć liczbę równocześnie aktywnych wątków.

2.3. Programowanie imperatywne

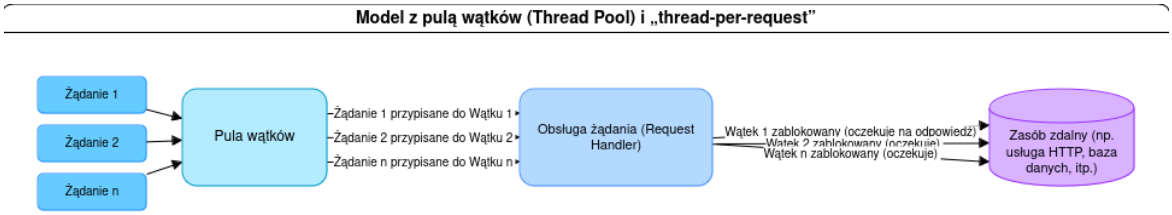
Programowanie imperatywne polega na opisywaniu kroków niezbędnych do osiągnięcia określonego celu. W tym paradygmacie nacisk kładziony jest na instrukcje zmieniające stan programu poprzez operacje na zmiennych. Programista definiuje szczegółowo, jak osiągnąć zamierzony wynik, korzystając z konstrukcji sterujących (pętli, instrukcji warunkowych) oraz przypisań. Aplikacja imperatywna jest to program, którego kod wykonuje się w sposób sekwencyjny. Oznacza to, że każda linia kodu może się wykonać jedynie po tym, jak poprzednia instrukcja została zakończona.

2.4. Tradycyjne wątki w Javie

Wątek systemowy to podstawowa jednostka wykonywania zadań zarządzana przez system operacyjny. Każdy wątek otrzymuje zasoby, takie jak pamięć stosu i czas procesora, co pozwala na niezależne wykonywanie operacji, ale sprawia, że nadmiar wątków

prowadzi do dużego narzutu systemowego. System operacyjny przydziela wątkom zasoby procesora za pomocą mechanizmu przełączania kontekstu, który zarządza czasem pracy procesora. Przełączanie to jednak wiąże się z kosztem, ponieważ wymaga zapisania stanu bieżącego wątku i załadowania stanu nowego. Tradycyjne aplikacje wielowątkowe często wykorzystują wątki systemowe do obsługi równoległych procesów.

Każdy wątek w Javie odpowiada bezpośrednio wątkowi systemowemu, co oznacza, że każde zadanie generuje wysokie zapotrzebowanie na zasoby, takich jak pamięć stosowa i czas na przełączanie kontekstu. Każdy wątek systemowy musi mieć z góry zarezerwowany stos o określonej wielkości w pamięci. Nawet jeżeli wątek nie wykorzystuje całego stosu, ta przestrzeń jest jednak zarezerwowana i niedostępna dla innych procesów czy wątków. Taki model działa dobrze w przypadku mniejszej liczby równoczesnych operacji, ale staje się ograniczeniem w środowiskach o wysokiej współbieżności, jak aplikacje mikroserwisowe. Problemem jest także ograniczona skalowalność, ponieważ liczba aktywnych wątków jest ograniczona przez dostępne zasoby, istnieje fizyczny limit liczby wątków. W aplikacjach korzystających z tradycyjnych wątków konieczne jest staranne planowanie puli wątków, aby uniknąć przeciążenia systemu.



Rysunek 2.2. Model puli wątków

2.4.1. „Thread per request" model

Jednym z najbardziej tradycyjnie stosowanych podejść w serwerach aplikacji do realizacji współbieżności jest model „jeden wątek na żądanie" (ang. *thread-per-request*). W tym podejściu każde przychodzące żądanie obsługiwane jest przez dedykowany wątek, który pozostaje związany z daną operacją od momentu jej rozpoczęcia do zakończenia. Aplikacja serwerowa startuje najczęściej z określoną pulą dostępnych wątków. Wątek taki, dopóki nie zostanie przydzielony do obsługi konkretnego żądania, pozostaje w stanie uśpienia (ang. *parked*), nie zużywając czasu procesora. Gdy pojawia się nowe żądanie, mechanizm zarządzania wątkami wybiera wolny wątek z puli, przydziela go do obsługi danego zadania, a po jego zakończeniu wątek wraca do puli, czekając na kolejne żądania. Taka architektura jest czytelna, intuicyjna i łatwa w utrzymaniu. Programista może myśleć o kodzie przetwarzającym żądanie w sposób sekwencyjny i imperatywny, co ułatwia zarówno proces tworzenia, jak i późniejszego debugowania aplikacji. Dodatkowo, profilowanie i rozpoznawanie źródeł problemów wydajnościowych jest stosunkowo proste, gdyż każdy wątek odpowiada dokładnie jednemu żądaniu. Jednak w miarę

skalowania systemu ograniczenia związane z wysokim kosztem utrzymania wielu wątków systemowych oraz dodatkowe obciążenia wynikające z blokujących operacji wątkowych stają się przeszkodą dla pełnego wykorzystania możliwości sprzętu.

2.5. Wątki wirtualne

Wątki wirtualne, wprowadzone w Javie 19, to lekkie wątki zarządzane na poziomie środowiska uruchomieniowego JVM [17], niezależnie od wątków systemowych. Wirtualny wątek nie jest powiązany z określonym wątkiem systemu operacyjnego. W przeciwieństwie do wątków platformy, ich tworzenie i obsługa są tańsze. Wirtualny wątek jest tworzony jako lekki obiekt na stercie Javy, a dopiero gdy faktycznie wykonuje kod, przypinany jest do jednego z niewielu fizycznych wątków systemowych. W efekcie można mieć tysiące czy nawet miliony wirtualnych wątków. Ich główną zaletą jest eliminacja narzutu wynikającego z przełączania kontekstu, ponieważ wiele wątków wirtualnych może działać na jednym wątku systemowym. Wątki wirtualne podczas wykonywania operacji blokujących nie zawieszają wątku platformy. Gdy wątek wirtualny wykonuje blokującą operację, środowisko uruchomieniowe Java zawiesza go tymczasowo, zapisuje jego stan na stercie i zwalnia powiązany wątek systemu operacyjnego do obsługi innych wątków wirtualnych [1]. Wątek wirtualny, w momencie odblokowania, może zostać przypięty do dowolnego innego wątku platformowego. Ułatwiają one implementację współbieżnych systemów, takich jak mikroserwisy, eliminując ograniczenia wynikające z działania tradycyjnych wątków.

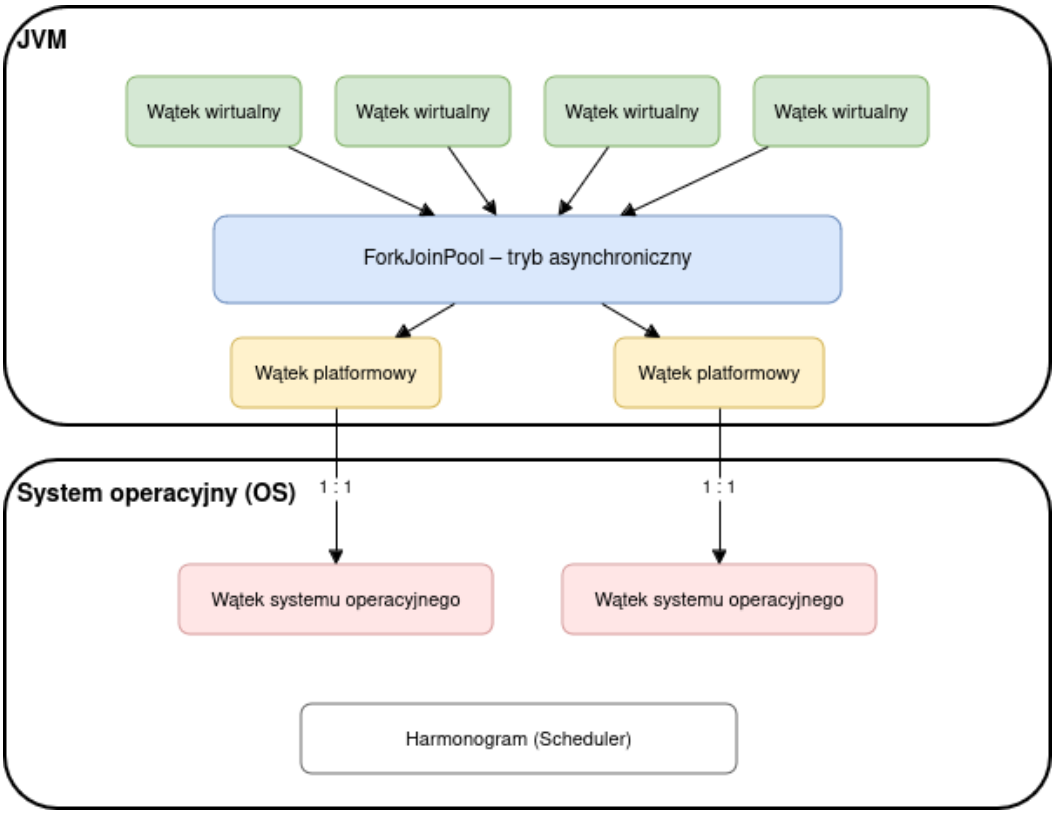
2.5.1. „Thread per request" bez puli wątków

W tradycyjnym modelu obsługi żądań, który opisuje się jako *Thread-Per-Request*, w praktyce wykorzystuje się pulę wątków systemowych o ograniczonej wielkości. Dzięki temu unikamy nadmiernego tworzenia i zwalniania wątków, co byłoby kosztowne przy dużej liczbie równoczesnych połączeń. Każde nowe żądanie jest przydzielane do wątku wyjętego z puli, a po zakończeniu pracy wątek trafia z powrotem do wolnych zasobów, gotowy do obsługi kolejnego zadania.

W podejściu z wątkami wirtualnymi koszty tworzenia i przełączania wątków są znacznie niższe, więc nie jest już konieczne ograniczanie się pulą niewielkiej liczby wątków. Każde żądanie może otrzymać osobny wątek wirtualny bez istotnego wpływu na wydajność. Dzięki temu uzyskujemy prostotę modelu *One-Thread-Per-Request* przy znacznie lepszej skalowalności, niż zapewniały klasyczne pule wątków systemowych.

2.5.2. Dynamiczne zarządzanie stosami

Tradycyjne wątki rezerwują podczas tworzenia z góry określony rozmiar stosu rzędu kilku megabajtów na wątek. Wątki wirtualne korzystają z dynamicznie przydzielanych stosów, co oznacza, że pamięć jest rezerwowana tylko wtedy, gdy jest potrzebna. W przeciwieństwie do zwykłych wątków, nie wymagają z góry zarezerwowanego stosu o



Rysunek 2.3. Diagram wątków wirtualnych i platformowych

określonej wielkości. Kiedy wątek wirtualny zostaje zawieszony, jego stan może zostać zapisany na stercie jako obiekt, dzięki czemu pamięć stosu zostaje częściowo zwolniona. Pozwala to uruchamiać dużo więcej wątków jednocześnie, przy niższym zapotrzebowaniu pamięciowym.

2.5.3. Kontynuacje

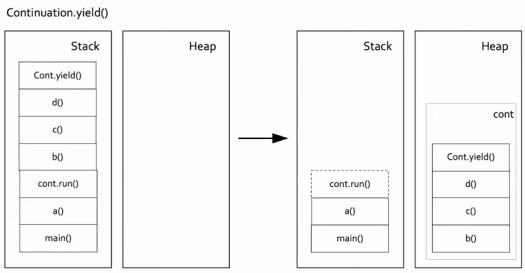
Kontynuacja (ang. Continuation) to struktura pozwalająca na zapamiętanie stanu wykonywanego kodu w określonym punkcie oraz wznowienie go w przyszłości dokładnie od tej samej instrukcji. W tradycyjnym modelu wywołań metod program działa w sposób liniowy i wątek podąża ciągiem instrukcji, aż do zakończenia lub przejścia do innego wątku. Kontynuacje wprowadzają jednak mechanizm, dzięki któremu można przerwać wykonanie kodu, zapisać jego stan, a następnie odtworzyć go z kontekstem stosu wywołań i wartościami lokalnymi w późniejszym momencie. Z punktu widzenia implementacji, każdy wątek wirtualny jest ściśle powiązany z obiektem *Continuation*. Taki obiekt przechowuje bieżący stan wykonywanego stosu. Gdy następuje zawieszenie, stan jest serializowany w *Continuation*, a pamięć związana z wątkiem wirtualnym może zostać przetworzona przez zarządzanie pamięcią JVM. Wznawianie wątku polega na odtworzeniu odpowiedniego stanu z *Continuation* w kolejnym dostępnym wątku platformowym.

Poniższy pseudokod ilustruje idee wznawiania i wstrzymywania wątków wirtualnych z wykorzystaniem koncepcji kontynuacji:

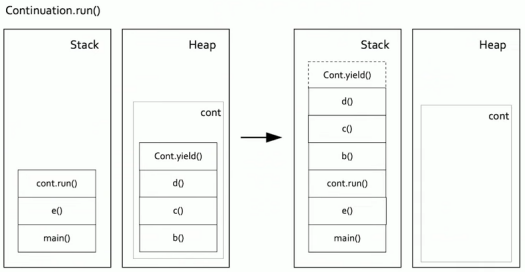
```
public static void main(String[] args) {
        // Tworzymy kontynuację z przypisanym zakresem
        Continuation continuation = new Continuation(SCOPE, () -> {
            System.out.println("[Continuation] Krok 1");
            // Zawieszenie kontynuacji
            Continuation.yield(SCOPE);

            System.out.println("[Continuation] Krok 2");
            // Kolejne zawieszenie
            Continuation.yield(SCOPE);

            System.out.println("[Continuation] Krok 3");
            // Po tej linii kontynuacja dobiegnie końca (isDone() == true).
        });

        // Dopóki kontynuacja nie została zakończona, wznawiamy jej wykonanie
        while (!continuation.isDone()) {
            continuation.run();
            System.out.println("[Main] Wróciliśmy do metody main po yield()");
        }

        System.out.println("[Main] Kontynuacja zakończona.");
    }
```

Listing 1. Kontynuacje

Kontynuacje nie są oficjalnie dostępne jako publiczny interfejs API, ponieważ jest to operacja niskiego poziomu. Powinny być używane wyłącznie przez autorów bibliotek do tworzenia interfejsów API wyższego poziomu, takich jak wątki wirtualne i inne wewnętrzne narzędzia języka Java.



Rysunek 2.4. Wstrzymanie



Rysunek 2.5. Wznowienie

2.5.4. ForkJoinPool jako domyślny planista

Wirtualne wątki w Javie są domyślnie obsługiwane przez pulę wątków systemowych obsługiwanych przez kolejkę FIFO (ang. first-in-first-out) typu `ForkJoinPool`, która działa jako planista (ang. scheduler) dla tych wątków, przyporządkowuje je do wolnych wątków systemowych. W sytuacji, gdy pewien wirtualny wątek blokuje się na operacji, ForkJoinPool może automatycznie dodać kolejne wątki systemowe do swojej puli, aby zrównoważyć niedostępność zablokowanych wątków. Gdy wątek wirtualny jest

tymczasowo przypięty i nie może wykonać innych zadań, pula rozszerza się, zapewniając, że kolejne wirtualne wątki wciąż mają zasoby do wykonywania swoich operacji. W ten sposób system minimalizuje negatywny wpływ blokujących wywołań na przepustowość i responsywność aplikacji opartej na wątkach wirtualnych. Zachowanie to można dostosować za pomocą parametrów systemowych:

- `jdk.virtualThreadScheduler.parallelism` – parametr ten określa bazowy poziom współbieżności, czyli ile platformowych wątków jest początkowo dostępnych do obsługi wątków wirtualnych. Wartość domyślna to liczba dostępnych procesorów zwrócona przez `Runtime.getRuntime().availableProcessors()`.
- `jdk.virtualThreadScheduler.maxPoolSize` – parametr ten kontroluje maksymalną wielkość puli wątków platformowych, które planista może wykorzystać w sytuacji, gdy część z nich zostanie zablokowana. Jeżeli ten parametr nie jest ustawiony, wartość domyślna wynosi 256.

2.5.5. Ograniczenia

Mechanizm `ThreadLocal` w językach takich jak Java pozwala na przechowywanie danych lokalnych dla wątku, nie współdzielonych z innymi wątkami. Dzięki niemu można unikać niepotrzebnej synchronizacji zarządzania obiektami pomiędzy wątkami. Jednak w przypadku wątków wirtualnych pojawiają się pewne dodatkowe wyzwania. Wątek wirtualny może być bardzo krótkotrwały, a więc dane przechowywane w `ThreadLocal` będą często tworzone i usuwane. Niewłaściwe zarządzanie tym mechanizmem może prowadzić do wycieków pamięci, szczególnie gdy referencje nie zostaną wyczyszczone w odpowiednim momencie, albo wzrostu ilości potrzebnej pamięci podczas uruchomienia wielu wirtualnych wątków jednocześnie.

Przypinanie wątków (ang. Pinning) to sytuacja, w której wątek wirtualny zostaje przypięty do konkretnego wątku platformowego. W praktyce oznacza to, że w momencie wykonywania blokującej operacji, wirtualny wątek jest przypinany do jednego konkretnego wątku systemu, aż do zakończenia tej operacji. Istnieją tylko dwa przypadki, w których wątek wirtualny jest przypięty do wątku systemowego:

- Gdy wykonuje kod wewnątrz zsynchronizowanego bloku lub metody (*synchronized*)
- gdy wywołuje natywną metodę lub obcą funkcję (wywołanie natywnej biblioteki przy użyciu JNI).

Po zakończeniu blokującej operacji pinning zostaje zwolniony, a wirtualny wątek powraca do swojej lekkiej natury, mogąc być dowolnie wykonywany przez dostępne wątki systemowe zarządzane przez środowisko. Przypinanie nie sprawia, że aplikacja działa niepoprawnie, ale może utrudniać jej skalowalność. Aby uniknąć problemów, konieczna

może być przebudowa kodu przez zastąpienie zsynchronizowanych metod/bloków przez inne metody synchronizacji, jak na przykład *ReentrantLock*.

2.6. Programowanie reaktywne

Programowanie reaktywne to paradygmat programowania, który skupia się na asynchronicznym i nieblokującym przetwarzaniu strumieni danych i reagowaniu na zdarzenia w czasie rzeczywistym. Programista definiuje, jak dane powinny być przetwarzane, a następnie biblioteka lub framework zajmuje się reakcją na zmiany danych i wywoływaniem odpowiednich operacji. W niniejszej sekcji omówiono podstawy programowania reaktywnego, jego kluczowe cechy oraz zastosowania w praktyce.

2.6.1. Podstawy programowania reaktywnego

W programowaniu reaktywnym dane są traktowane jako strumienie, które mogą być obserwowane i przetwarzane jako ciąg zdarzeń w czasie rzeczywistym. Podejście to opiera się na idei tworzenia oprogramowania, które reaguje na zmiany w środowisku, a nie jest aktywnie kontrolowane przez programistę. Każda zmiana w strumieniu inicjuje reakcję, która jest obsługiwana w sposób asynchroniczny. Innymi słowy, jest to programowanie z asynchronicznymi strumieniami danych, które przesyłają dane do konsumenta, gdy stają się one dostępne, umożliwiając programistom projektowanie kodu, który reaguje szybko i asynchronicznie.

Kluczową rolę odgrywa tutaj model *publish-subscribe*, który oddziela producentów danych od ich konsumentów. Dzięki temu komunikacja między komponentami jest luźno powiązana, co sprzyja skalowalności i elastyczności aplikacji. Każde przetworzenie danych możena nazwać zdarzeniem, które przekazywane jest pomiędzy publisherem, który odpowiada za publikowanie, a subscriberem odpowiadającym za nasłuchiwanie i odczytywanie danych.

Asynchroniczność oznacza, że operacje w programowaniu reaktywnym nie blokują wątków na czas oczekiwania na wynik. Zamiast tego, zadania są rejestrowane jako obietnice (ang. promises) lub przepływy reaktywne (ang. reactive streams), które zostaną obsłużone, gdy dane będą dostępne. Takie podejście minimalizuje użycie zasobów, co czyni je szczególnie efektywnym w środowiskach o dużym obciążeniu.

2.6.2. Callback

Procedura zwrotna (ang. callback) to funkcja, którą przekazuje się do innego komponentu, aby została wywołana w przyszłości w momencie wystąpienia określonego zdarzenia. W przeciwieństwie do wywołań synchronicznych, gdzie parametry są przekazywane do procedury, a przepływ programu zostaje wstrzymany do momentu zwrócenia wyniku, w komunikacji asynchronicznej wywołujący nie oczekuje na rezultat bezpośrednio. Zamiast tego rejestruje procedurę zwrotną, która zostanie wywołana, gdy wynik będzie dostępny. Pozwala to wywołującemu kontynuować pracę bez blokowania

przepływu programu, a procedura zwrotna zostanie uruchomiona w odpowiednim momencie. Innymi słowy, wywołujący przekazuje do procedury asynchronicznej informację o tym, co powinno się wydarzyć w przyszłości, gdy wynik będzie możliwy do wyliczenia.

Dodatkowo, wywołujący może zarejestrować wiele procedur zwrotnych. Jest to przydatne, gdy chce zostać powiadomiony nie tylko o pomyślnym zakończeniu operacji, ale także o ewentualnych błędach. W takim przypadku możliwe jest zarejestrowanie oddzielnych procedur dla sukcesu i dla awarii. Co więcej, wywołujący sam może stać się wywoływanym, przetwarzając otrzymany wynik i przekazując go dalej w łańcuchu przetwarzania.

2.6.3. Wzorzec obserwatora

Wzorzec *callback* jest często implementowany za pomocą wzorca obserwatora. W tym wzorcu wartość zwracana przez asynchroniczne wywołanie procedury nazywana jest *Obserwowanym* (*ang. Observable*), natomiast procedura zwrotna to *Obserwator* (*ang. Observer*).

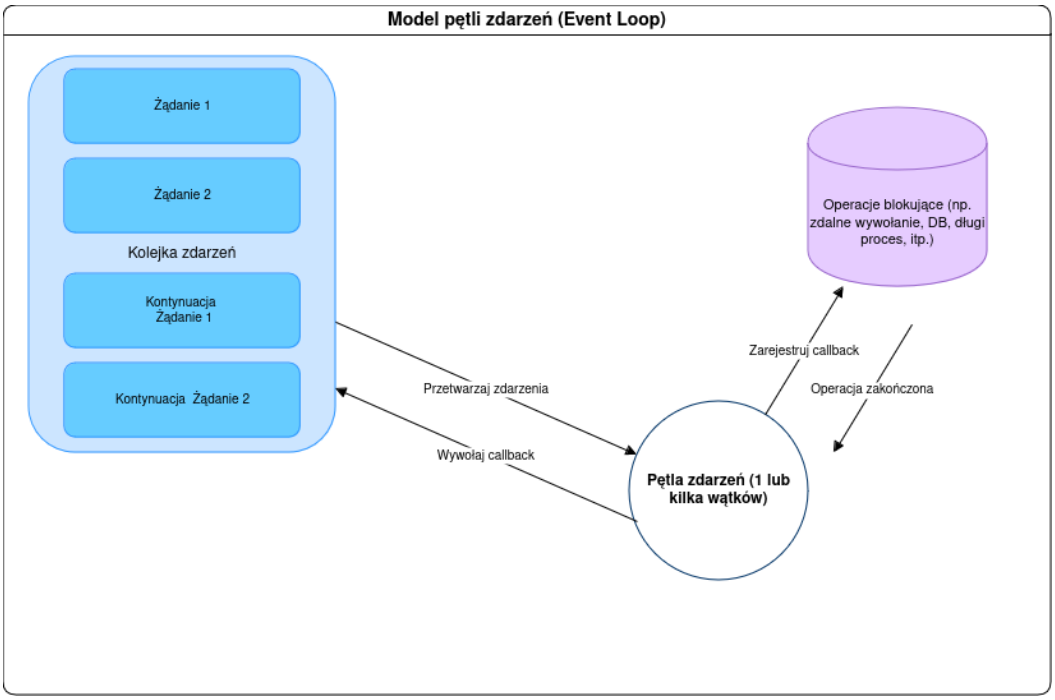
Przykładem zastosowania wzorca obserwatora jest model *publish-subscribe*, w którym serwisy publikują zdarzenia, a subskrybenci reagują na nie zgodnie ze swoimi potrzebami. Polega on na mechanizmie, w którym jeden obiekt, zwany *wydawcą* (*ang. publisher*), powiadamia zarejestrowane obiekty *obserwatorów* (*ang. subscribers*) o zmianach w swoim stanie. Dzięki temu obserwatorzy mogą reagować na te zmiany w sposób asynchroniczny i nieblokujący.

Główną zaletą wzorca obserwatora jest luźne powiązanie między wydawcą a obserwatorami. Wydawca nie musi znać szczegółów implementacyjnych obserwatorów, co czyni system bardziej elastycznym i skalowalnym. Obserwatorzy mogą być dynamicznie dodawani lub usuwani, co pozwala na łatwe dostosowanie systemu do zmieniających się wymagań.

2.6.4. Event loop

Pętla zdarzeń (ang. event loop) to mechanizm, który nasłuchuje na pewne zdarzenia asynchroniczne (np. gotowość kanału I/O, sygnał zakończenia, timer) i wywołuje odpowiednie callbacki, gdy te zdarzenia nastąpią. Jest to mechanizm, który monitoruje kolejkę zdarzeń i reaguje na nie w sposób asynchroniczny. Dzięki temu logika aplikacji może być ustrukturyzowana wokół zdarzeń takich jak zakończenie operacji I/O czy nadejście danych, a nie wokół sztywnych sekwencji kroków.

Pętla działa w jednym, głównym wątku i nieustannie monitoruje kolejkę zdarzeń, takich jak zakończenie operacji wejścia-wyjścia, nadejście danych z zewnętrznych źródeł czy sygnały wywołane wewnątrz systemu. Gdy pojawia się nowe zdarzenie, *event loop* wywołuje odpowiedni fragment kodu (tzw. *callback*) przypisany do danego zdarzenia. Kluczowe jest tutaj asynchroniczne, nieblokujące podejście: operacje, które w podejściu



Rysunek 2.6. Model pętli zdarzeń (Event Loop).

imperatywnym oczekiwałyby na zakończenie (np. czasochłonny dostęp do bazy danych), w podejściu reaktywnym rejestrują swój callback i natychmiast zwalniają event loop do obsługi kolejnych zdarzeń. W ten sposób cały program może być napisany w sposób reagujący na zdarzenia, bez konieczności blokowania głównego wątku na długotrwające operacje. To prowadzi do wysoce skalowalnych rozwiązań, które mogą obsługiwać ogromną liczbę konkurencyjnych połączeń sieciowych, minimalizując narzut związany z kontekstem wątków czy koniecznością zastosowania skomplikowanych mechanizmów synchronizacji.

2.6.5. Cechy systemów reaktywnych

Główne cechy programowania reaktywnego zostały opisane w dokumencie „Reactive Manifesto" [23]. Są to:

- **Responsywność:** Systemy reaktywne zapewniają szybkie reagowanie na zdarzenia, nawet przy dużym obciążeniu.
- **Elastyczność:** Dzięki luźnemu powiązaniu komponentów, aplikacje mogą być łatwo skalowane wertykalnie i dostosowywane do zmieniających się wymagań.
- **Odporność:** Systemy są zaprojektowane w taki sposób, aby radzić sobie z błędami, izolując awarie i zapobiegając propagacji problemów.
- **Skalowalność:** Programowanie reaktywne umożliwia efektywne zarządzanie zasobami, co pozwala na obsługę dużej liczby równoczesnych żądań.

Te cechy sprawiają, że podejście reaktywne jest szczególnie atrakcyjne w aplikacjach, w których kluczowa jest współbieżność i efektywne wykorzystanie zasobów. Typowe

przypadki użycia to aplikacje webowe, w których serwer musi obsłużyć wiele równoczesnych żądań. Dzięki asynchronicznemu modelowi obsługi aplikacje mogą przyjmować więcej połączeń przy mniejszym zużyciu wątków systemowych.

2.6.6. Reaktywność w Javie

W świecie Javy popularnymi bibliotekami wspierającymi programowanie reaktywne są *RxJava* i *Project Reactor*. Obie umożliwiają zarządzanie strumieniami danych, oferując bogaty zestaw operatorów do przekształcania, filtrowania i łączenia danych. W kontekście mikroserwisów programowanie reaktywne pozwala na budowanie systemów zdolnych do efektywnej obsługi rozproszonych komponentów.

Przykładowo, w aplikacjach opartych na Spring WebFlux, programowanie reaktywne umożliwia tworzenie nieblokujących kontrolerów HTTP, które mogą obsłużyć tysiące równoczesnych żądań. Podobnie w systemach IoT podejście reaktywne pozwala na dynamiczne reagowanie na zdarzenia generowane przez sensory. Dzięki swoim zaletom, programowanie reaktywne staje się standardem w projektowaniu skalowalnych i wydajnych aplikacji.

Podsumowując, w codziennej pracy z kodem reaktywnym nie ma potrzeby ręcznie tworzyć *event loop* czy zarządzać *callbackami* w klasycznym stylu, ponieważ jest to obsługiwane przez gotowe biblioteki. Cały model reaktywny to w zasadzie rozwinięcie wzorca obserwatora wspieranego asynchronicznymi pętlami zdarzeń i callbackami.

Tabela 2.1. Porównanie Reactor WebFlux (Reaktywny) i Virtual Threads (Imperatywny)

Aspekt	Reactor WebFlux (Reaktywny)	Virtual Threads (Imperatywny)
Paradygmat programowania	Reaktywny, oparty na zdarzeniach i strumieniach	Imperatywny, tradycyjny model blokujący
Model współbieżności	Zarządzanie współbieżnością przez strumienie reaktywne (`Flux`, `Mono`)	Zarządzanie współbieżnością przez lekkie wątki wirtualne
Zarządzanie wątkami	Wykorzystuje pętlę zdarzeń (event loop) i niewielką liczbę wątków	JVM zarządza wątkami wirtualnymi, umożliwiając tworzenie tysięcy lekkich wątków
Skalowalność	Bardzo wysoka dla zadań I/O – obsługa dużej liczby równoczesnych połączeń i żądań	Wysoka skalowalność zarówno dla zadań I/O, jak i CPU (*1 żądanie – 1 wątek wirtualny*)
Łatwość użycia	Wyższy poziom skomplikowania, wymaga zrozumienia podejścia reaktywnego	Bardziej zbliżone do tradycyjnego podejścia wątkowego, mniejszy poziom skomplikowania
Obsługa back-pressure	Wbudowana w specyfikację Reactive Streams, ułatwia sterowanie przepływem danych	Brak wbudowanych mechanizmów back-pressure; trzeba stosować tradycyjne ograniczenia lub kolejki
Wydajność	Zoptymalizowana pod kątem operacji I/O z niskim narzutem na zarządzanie wątkami	Bardzo dobra zarówno dla operacji I/O, jak i CPU, dzięki lekkości wątków
Integracja z istniejącym kodem	Wymaga większych zmian w kodzie (przejście na asynchroniczne API, strumienie `Flux/Mono`)	Łatwe wprowadzenie do istniejącego kodu; podobieństwo do `Thread` w Javie
Obsługa błędów	Bardziej złożona; propagacja błędów w strumieniach reaktywnych (`onErrorResume`, itp.)	Prosta, podobna do klasycznego modelu z wyjątkami
Przypadki użycia	Mikroserwisy, streaming danych, aplikacje oparte na intensywnych operacjach I/O	Aplikacje wymagające wysokiej współbieżności (I/O i CPU), z zachowaniem prostoty kodu
Testowanie i debugowanie	Trudniejsze przez asynchroniczność i tzw. *operator chaining*	Prostsze - zbliżone do debugowania tradycyjnych wątków
Monitorowanie / Observability	Wymaga narzędzi wspierających reaktywne strumienie (np. Reactor, RSocket)	Można używać standardowych narzędzi do profilowania wątków i stosu wywołań
Zarządzanie kontekstem	Potrzebuje dedykowanych mechanizmów (np. `Reactor Context`) do propagacji kontekstu	Kontekst można przekazywać tak jak w klasycznych wątkach, mniejsza złożoność

2.7. Testowanie

Testowanie jest etapem procesu tworzenia oprogramowania, pozwalającym na ocenę jego jakości, wykrycie błędów oraz zrozumienie zachowania aplikacji w różnych warunkach. Poniżej przedstawiono główne aspekty teorii testowania, które można uwzględnić w badaniu [6]:

2.7.1. Cele testowania

Testowanie oprogramowania ma na celu:

- **Weryfikację poprawności działania**: Sprawdzenie, czy aplikacja spełnia założenia projektowe.
- **Identyfikację problemów wydajnościowych**: Ujawnienie ograniczeń związanych z czasem odpowiedzi, użyciem zasobów czy skalowalnością.
- **Porównanie alternatywnych podejść**: Ocena różnic między różnymi technologiami lub architekturami, np. wirtualnymi wątkami i programowaniem reaktywnym.

W kontekście tej pracy badawczej celem testowania jest porównanie efektywności dwóch podejść do współbieżności w Javie, co wymaga zarówno odpowiedniego planu, jak i odpowiednich narzędzi testowych.

2.7.2. Rodzaje testów

Testy funkcjonalne Testy funkcjonalne skupiają się na poprawności działania aplikacji zgodnie z jej wymaganiami. W przypadku aplikacji opartych na wirtualnych wątkach i programowaniu reaktywnym należy sprawdzić:

- Czy żądania użytkowników są poprawnie obsługiwane.
- Jak aplikacja reaguje na różne scenariusze wejściowe.

Testy wydajnościowe Testy wydajnościowe są kluczowe w tej pracy, ponieważ umożliwiają ocenę takich metryk, jak:

- Czas odpowiedzi
- Wykorzystanie zasobów
- Skalowalność

Testy obciążeniowe i stresowe

- **Testy obciążeniowe**: Określają, jak aplikacja działa pod maksymalnym oczekiwanym obciążeniem[14].
- **Testy stresowe**: Badają, jak system reaguje w warunkach ekstremalnych, np. przy większej liczbie żądań niż zakładana w projekcie.

Testy porównawcze Porównanie dwóch technologii wymaga testów o identycznych warunkach wejściowych. Przykładowo:

- Obciążenie o tej samej charakterystyce.
- Jednolite środowisko testowe.

2.7.3. Metryki wydajnościowe

Wydajność aplikacji webowych mierzy się, podając jej określone obciążenie i obserwując takie parametry jak *opóźnienie* (ang. latency), *przepustowość* (ang. throughput) oraz *zużycie zasobów* (CPU, pamięć). Istotne jest zminimalizowanie czynników zakłócających, takich jak dodatkowe procesy czy zmienne środowiskowe, aby wyniki pomiarów nie były zaburzone.

Metryki wydajnościowe dostarczają danych do analizy wyników testów. Brendan Gregg [12] wyróżnia trzy główne metryki dla środowisk aplikacji webowych:

- **Throughput (przepustowość)**: Liczba żądań przetworzonych w jednostce czasu.
- **Latency (opóźnienie)**: Czas odpowiedzi aplikacji na pojedyncze żądanie.
- **Użycie zasobów**: informuje o wykorzystaniu CPU i pamięci

Zarówno monitorowanie opóźnienia [28], jak i przepustowości ukazuje, czy system skaluje się poprawnie wraz ze wzrostem obciążenia. Natomiast analiza zużycia zasobów umożliwia identyfikację momentu, w którym aplikacja osiąga granice wydajności i nie jest w stanie przetwarzać żądań szybciej. Dzięki zrozumieniu tych parametrów można odpowiednio dostosować konfigurację oraz wykryć potencjalne źródła problemów.

3. Środowisko eksperymentalne

3.1. Specyfikacja środowiska testowego

Środowisko testowe zostało uruchomione na komputerze o następującej specyfikacji sprzętowej i programowej:

- **System operacyjny:** Ubuntu 20.04.6 LTS (Focal Fossa)
- **Procesor:** 12th Gen Intel(R) Core(TM) i7-1265U, 10 rdzeni, 12 wątków, taktowanie bazowe 2.7 GHz, maksymalne 4.8 GHz.
- **Pamięć RAM:** 32GB LPDDR5 5200 MHz
- **Dysk:** Dysk NVMe 953.9 GiB
- **Wirtualizacja:** VT-x.
- **Biblioteki:** Spring Boot Parent w wersji 3.4
- **Platforma Java:** Java 21
- **Baza danych:** PostgreSQL 17

3.1.1. Izolacja środowiska testowego

Aby zminimalizować wpływ czynników zewnętrznych, które mogą wpłynąć na wynik, zastosowano odpowiednie strategie izolacji:

1. **Dedykowane środowisko testowe**:
    - Użycie oddzielnych maszyn fizycznych lub wirtualnych, przeznaczonych wyłącznie do testów.
    - Wyłączenie zbędnych procesów i usług w systemie operacyjnym.
2. **Konteneryzacja**:
    - Wykorzystanie technologii takich jak Docker do uruchamiania aplikacji w odizolowanym środowisku, co zapewnia spójność konfiguracji.
3. **Stabilność zasobów sprzętowych**:
    - Przydzielenie stałej ilości zasobów (CPU, pamięci, I/O) aplikacjom testowanym, aby uniknąć rywalizacji o zasoby.
4. **Replikowalna konfiguracja środowiska**:
    - Automatyzacja procesu konfiguracji środowiska testowego za pomocą Docker Compose

Spójne środowisko testowe pozwala wyeliminować czynniki zakłócające, dzięki czemu możliwe będzie uzyskanie równomiernych i powtarzalnych wyników, koncentrując się wyłącznie na różnicach wynikających z używanych technologii.

3.2. Zastosowane technologie

3.2.1. Spring Framework

Istnieje wiele możliwych rozwiązań do tworzenia usług webowych w Javie. Jednym z takich rozwiązań jest framework Spring [25], który jest open-source'owym frameworkiem będącym częścią ekosystemu Javy.

Spring Boot Spring Boot to framework oparty na Spring Framework, umożliwiający szybsze tworzenie usług webowych dzięki trzem kluczowym funkcjom: automatycznej konfiguracji, narzuconemu podejściu (*ang. opinionated approach*) oraz możliwości tworzenia aplikacji samodzielnych (*ang. standalone applications*). Dzięki automatycznej konfiguracji i pakietom startowym, znacząco zmniejsza się liczba konfiguracji koniecznych do uruchomienia w pełni funkcjonalnej usługi webowej. Dodatkowo, użytkownik ma możliwość selektywnego doboru modułów, co pozwala na wygodną personalizację rozwiązań w zależności od potrzeb.

Spring WebMVC *Spring WebMVC* to tradycyjny, oparty na wzorcu *Model-View-Controller* moduł Spring, przeznaczony do tworzenia blokujących usług webowych. Daje on możliwość łatwej implementacji kontrolerów, widoków i modeli, co przyspiesza proces tworzenia aplikacji [27], [29]. Rozwiązanie to jest dobrze znane, łączy w sobie wszystkie zalety wzorca MVC z wygodą Spring.

WebFlux *WebFlux* to moduł Spring zorientowany na programowanie reaktywne, zbudowany na bibliotece *Project Reactor* [26]. Wykorzystuje on specyfikację *Reactive Streams*, implementując wzorzec wydawcy i subskrybenta [30]. Rozwiązanie to umożliwia asynchroniczną, nieblokującą obsługę żądań, dzięki czemu można budować skalowalne systemy zdolne do obsługi dużej liczby zapytań. W modelu reaktywnym subskrybenci reagują na zdarzenia od wydawców, co pozytywnie wpływa na wydajność aplikacji oraz pozwala efektywniej wykorzystać zasoby sprzętowe.

Spring Boot Actuator *Spring Boot Actuator* to rozszerzenie umożliwiające monitorowanie oraz zarządzanie uruchomioną aplikacją. Zapewnia ono szereg wbudowanych *endpointów*, dzięki którym można uzyskać informacje na temat metryk, statusu dostępności aplikacji czy konfiguracji. Funkcjonalności te pozwalają na szybką diagnostykę potencjalnych problemów, a także ułatwiają optymalizację i utrzymanie środowiska produkcyjnego.

Serwery aplikacji W badaniach wykorzystano domyślne serwery aplikacji – *Tomcat* dla środowiska *Spring WebMVC* oraz *Netty* dla aplikacji opartych na *Spring WebFlux*. Pozwoliło to na zachowanie standardowych, wspieranych przez platformę konfiguracji, bez wprowadzania dodatkowych modyfikacji.

3.2.2. Docker

W celu zapewnienia izolacji środowisk oraz uzyskania powtarzalnych i miarodajnych wyników testów wydajnościowych, aplikacje uruchamiane były w kontenerach *Docker* [7]. Takie podejście pozwala na odseparowanie testowanego oprogramowania od konfiguracji lokalnego systemu, zapewniając kontrolowane, powtarzalne środowisko uruchomieniowe, które minimalizuje wpływ czynników zewnętrznych na wyniki pomiarów.

Do implementacji środowiska wykorzystano oficjalny obraz systemowy *openjdk:23-jdk-slim*, który oferuje zredukowaną do minimum, a zarazem optymalną pod względem wydajności bazę do uruchamiania aplikacji języka Java. W celu standaryzacji zasobów, każdemu kontenerowi przydzielono następujące limity:

- Pamięć operacyjna: 2 GB RAM
- Zasoby procesora: maksymalnie dostępne jednostki CPU hosta

Taka konfiguracja gwarantuje, że poszczególne kontenery uruchamiają aplikację w niemal identycznych warunkach, co ułatwia porównywanie wyników między poszczególnymi iteracjami testów oraz między różnymi wariantami optymalizacji [19].

3.3. Narzędzia monitorujące: Prometheus i Grafana

W celu badania wydajności aplikacji wykorzystano narzędzia do monitorowania oraz wizualizacji danych Prometheus i Grafana.

3.3.1. Prometheus

Prometheus [3] to otwartoźródłowy system monitorowania, opracowany pierwotnie w firmie SoundCloud, a następnie rozwijany przez społeczność. Jest on obecnie jednym z najpopularniejszych narzędzi do monitorowania usług chmurowych i mikroserwisów. Prometheus opiera się na modelu *pull*, w którym serwer cyklicznie pobiera dane z określonych źródeł. Aplikacje mierzone przez Prometheusa wystawiają odpowiednie endpointy, z których pobierane są informacje. Dane przechowywane są w bazie TSDB, umożliwiającej ich efektywne składowanie oraz szybki odczyt. Oferowany przez Prometheusa język zapytań PromQL pozwala na przetwarzanie i agregowanie danych, a także na tworzenie złożonych analiz. System łatwo integruje się z narzędziami wizualizacyjnymi.

3.3.2. Grafana

Grafana [10] to otwartoźródłowa platforma służąca do wizualizacji danych pochodzących z różnych źródeł, w tym z Prometheusa. Umożliwia ona tworzenie wykresów prezentujących dane w czytelnej, graficznej formie. Pozwala ona na łączenie danych z różnych źródeł w jednym miejscu.

3.4. Narzędzia testowe

W celu oceny wydajności oraz skalowalności badanych rozwiązań zastosowano narzędzie do testów obciążeniowych *k6* [16]. Jest to oprogramowanie służące do symulowania ruchu użytkowników i pomiaru kluczowych metryk wydajności. *k6* charakteryzuje się prostotą integracji, elastycznością konfigurowania [15].

Kluczowe cechy platformy *k6* to:

- Język skryptów testowych: Scenariusze testowe definiowane są w JavaScript, co upraszcza proces pisania i modyfikowania testów.
- Lekka i wydajna architektura: *k6* zostało zaprojektowane w języku Go, dzięki czemu cechuje się wysoką wydajnością przy stosunkowo niewielkim zużyciu zasobów. Pozwala to na generowanie obciążenia o dużym wolumenie przy zachowaniu stabilności i przewidywalnych czasów odpowiedzi.
- Obszerne raportowanie i wizualizacja wyników: *k6* umożliwia eksport wyników w formacie JSON, a także integrację z dedykowanymi narzędziami do wizualizacji (np. Grafana). Dzięki temu możliwe jest przejrzyste przedstawienie wyników oraz ich dokładna analiza.

W ramach przeprowadzonych badań, *k6* posłużyło do odtworzenia realistycznych scenariuszy obciążenia, w tym ruchu generowanego przez wielu równocześnie działających użytkowników. Uzyskane dane, takie jak średni czas odpowiedzi, wskaźniki błędów czy maksymalne wartości opóźnień, umożliwiły dokonanie adekwatnej oceny skalowalności rozwiązań, identyfikację wąskich gardeł oraz sprawdzenie efektywności zastosowanych metod optymalizacyjnych.

4. Metodologia

Aby zapewnić wiarygodność i powtarzalność wyników, każdy test został poprzedzony fazą rozgrzewkową. Pozwala to na ustabilizowanie środowiska wykonawczego oraz uniknięcie wpływu czynników zewnętrznych na pomiary. Po rozgrzewce przeprowadzono właściwe testy, dla różnych poziomów współbieżności: 1000 oraz 5000 równoczesnych połączeń.

1. **Rozgrzewka (ang. warmup)**: Stopniowe zwiększanie liczby wirtualnych użytkowników (VU) do 1000 w ciągu 15 sekund, pozwalające na stabilizację systemu przed właściwym testem.
2. **Stałe obciążenie (ang. steady load)**: Utrzymanie stałej liczby VU przez 2 minuty i 30 sekund, symulujące rzeczywiste obciążenie systemu.
3. **Schładzanie (ang. rampdown)**: Stopniowe zmniejszanie liczby VU do zera w ciągu 15 sekund, kończące test.

4.1. Badane metryki wydajności i zasobów

Poniżej omówiono każdą grupę metryk wykorzystanych w celu dokładnej oceny wydajności badanych aplikacji oraz ich znaczenie [8]:

- **Metryki związane z obsługą żądań HTTP:**
  - *Nieudane żądania (%)*: Procentowa liczba żądań zakończonych błędem.
  - *Liczba żądań na sekundę*: Średnia liczba obsługiwanych żądań HTTP na sekundę. Jest to miara przepustowości aplikacji, pokazująca, ile żądań jest w stanie efektywnie przetworzyć w danym czasie.
- **Metryki czasu obsługi żądań (w milisekundach):**
  - *Minimalny czas*: Najkrótszy zmierzony czas odpowiedzi.
  - *Maksymalny czas*: Najdłuższy zmierzony czas odpowiedzi.
  - *Średni czas*: Średni czas obsługi żądań.
  - *Mediana*: Czas odpowiedzi pośrodku zbioru próbek, gdzie połowa żądań jest szybsza, a połowa wolniejsza.
  - *P90 i P95*: Czasy graniczne, poniżej których mieści się odpowiednio 90% i 95% wszystkich żądań. Wskaźniki te pomagają ocenić jakość obsługi w gorszych, lecz wciąż częstych przypadkach, lepiej charakteryzując doświadczenie większości użytkowników niż sama średnia.
- **Metryki CPU:**
  - *Średnie i maksymalne użycie CPU kontenera*: Określają, w jakim stopniu zasoby CPU maszyny są wykorzystywane przez dany kontener.
  - *Średnie i maksymalne użycie CPU systemu JVM*: Pokazują, jaka część dostępnej mocy obliczeniowej jest zużywana przez środowisko uruchomieniowe JVM.

  - — *Średnie i maksymalne użycie CPU procesu JVM*: Odnoszą się bezpośrednio do procesu aplikacji działającego w JVM, pozwalając zrozumieć, jak duże jest obciążenie procesora generowane przez kod aplikacji.
- **Metryki pamięci kontenera i JVM:**
  - — *Średnia i maksymalna użyta pamięć kontenera*: Informują o poziomie wykorzystania pamięci RAM przez kontener (cały proces, wliczając w to JVM i narzut środowiska).
  - — *Średnia i maksymalna użyta pamięć JVM*: Odnosi się do zasobów pamięci używanych wewnątrz JVM (m.in. heap i non-heap).
- **Metryka wątków:**
  - — *Maksymalna liczba wątków*: Określa największą liczbę aktywnych wątków w JVM.
- **Metryki obciążenia systemu (ang. Load Average):**
  - — *Średnie i maksymalne obciążenie systemu*: Wartości te wskazują odpowiednio średnią i maksymalną liczbę zadań oczekujących na dostęp do zasobów CPU w danym przedziale czasu.

4.2. Odchylenie standardowe wyników testów

W celu oszacowania stabilności i powtarzalności wyników przeprowadzono pięciokrotne uruchomienie testów wydajnościowych. Odchylenie standardowe zostało obliczone w celu zbadania poziomu zróżnicowania wyników w poszczególnych iteracjach.

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}, \tag{1}$$

gdzie:

- x_i to wynik pojedynczego testu,
- $\bar{x}$ to średnia wartość wyników testów,
- n to liczba przeprowadzonych testów.

Wyniki analizy wykazały, że odchylenie standardowe wynosiło 11%, co oznacza umiarkowany poziom rozbieżności w uzyskanych pomiarach. Pomimo niewielkich różnic pomiędzy kolejnymi uruchomieniami, wartości te pozostają w akceptowalnych granicach tolerancji dla tego typu testów.

Uzyskany poziom odchylenia standardowego może wynikać z czynników środowiskowych, takich jak obciążenie systemu w trakcie testów, oraz wpływu innych procesów systemowych.

4.3. Warianty konfiguracji aplikacji WebMVC

Na potrzeby przeprowadzonych badań aplikacja WebMVC została skonfigurowana w trzech wariantach, różniących się sposobem zarządzania wątkami w serwerze Tomcat. Zmiany te zostały wprowadzone w plikach konfiguracyjnych aplikacji Spring Boot, takich jak `application.properties` lub `application.yml`. Dla wariantów **web-mvc-app** oraz **web-mvc-app-2** modyfikacje dotyczyły głównie parametru `server.tomcat.max-threads`, natomiast dla wariantu **web-mvc-vt-app** dodatkowo skonfigurowano parametr `spring.virtual-threads.enabled`, co umożliwiło wykorzystanie wątków wirtualnych.

1. **web-mvc-app** - liczba wątków serwera Tomcat została ustawiona na wartość domyślną, wynoszącą **200**. Jest to standardowa konfiguracja stosowana w wielu środowiskach produkcyjnych.
2. **web-mvc-app-2** - liczba wątków serwera Tomcat została zwiększona do **1000**. Pozwoliło to zbadać wydajność aplikacji przy większej liczbie jednoczesnych żądań obsługiwanych przez klasyczne, systemowe wątki.
    - `server.tomcat.max-threads` został ustawiony na **1000**, co zwiększa pulę wątków serwera Tomcat.
3. **web-mvc-vt-app** - w tej konfiguracji włączono obsługę **wątków wirtualnych**,
    - `spring.virtual-threads.enabled` został ustawiony na **true**, co włącza obsługę wątków wirtualnych w aplikacji Spring.

5. Eksperymenty

W eksperymentach wykorzystano cztery scenariusze testowe, zróżnicowane pod względem charakteru wykonywanych operacji:

5.1. Scenariusz 1: Hello World

5.1.1. Opis

Pierwszy scenariusz polega na obsłudze żądania HTTP, które zwraca prosty łańcuch znaków "Hello World". Ten test ma na celu ocenę wydajności serwera w sytuacji minimalnego obciążenia logicznego.

5.1.2. Wyniki

Tabela 5.1. Porównanie wydajności API api-hello dla scenariusza 1000 użytkowników

	web-mvc-app	web-mvc-app-2	web-mvc-vt-app	webflux-app
Zadania HTTP				
Nieudane żądania (%)	0.00%	0.00%	0.00%	0.00%
Liczba żądań na sekundę	20068.89	17213.57	20845.16	22061.38
Czas żądania (ms)				
Minimalny czas	0.56	0.32	0.24	0.23
Maksymalny czas	2400.0	3550.0	2540.0	555.57
Średni czas	47.66	52.75	43.01	40.54
Mediana	35.75	38.31	34.14	31.74
P90	95.2	115.81	74.17	67.33
P95	129.43	138.59	109.6	99.87
Użycie CPU (%)				
Średnie użycie CPU	41.11	41.77	38.64	46.73
Maksymalne użycie CPU	48.15	44.86	40.22	55.58
Średnie użycie CPU systemu JVM	57.51	59.69	55.47	64.24
Maksymalne użycie CPU systemu JVM	75.86	77.63	80.85	90.26
Średnie użycie CPU procesu JVM	53.63	55.74	51.93	68.49
Maksymalne użycie CPU procesu JVM	80.05	80.72	88.17	100.0
Pamięć kontenera (MiB)				
Średnia użyta pamięć	543.62	666.26	569.53	538.48
Maksymalna użyta pamięć	572.67	711.31	608.04	557.68
Pamięć JVM (MiB)				
Średnia użyta pamięć JVM	237.48	265.96	269.64	166.25
Maksymalna użyta pamięć JVM	348.74	385.99	387.12	230.93
Wątki (Threads)				
Maksymalna liczba wątków	214.0	1014.0	31.0	38.0
Obciążenie (Load Average)				
Średnie obciążenie	16.51	17.08	13.15	12.14
Maksymalne obciążenie	24.62	21.4	18.55	16.67

Tabela 5.2. Porównanie wydajności API api-hello dla scenariusza 5000 użytkowników

	web-mvc-app	web-mvc-app-2	web-mvc-vt-app	webflux-app
Zadania HTTP				
Nieudane żądania (%)	0.03%	0.03%	0.02%	0.00%
Liczba żądań na sekundę	17503.88	16530.43	19234.02	19703.5
Czas żądania (ms)				
Minimalny czas	0.92	0.73	0.73	0.62
Maksymalny czas	60000.0	60000.0	60000.0	2250.0
Średni czas	222.88	240.94	205.72	215.5
Mediana	138.48	151.86	126.36	159.25
P90	448.26	474.14	372.24	398.9
P95	573.18	587.3	471.63	604.18
Użycie CPU (%)				
Średnie użycie CPU	48.07	47.57	34.2	50.99
Maksymalne użycie CPU	49.76	52.06	47.61	52.41
Średnie użycie CPU systemu JVM	67.75	66.11	66.6	72.51
Maksymalne użycie CPU systemu JVM	92.79	93.67	90.2	91.77
Średnie użycie CPU procesu JVM	66.89	64.87	67.03	83.43
Maksymalne użycie CPU procesu JVM	96.49	97.99	100.0	100.0
Pamięć kontenera (MiB)				
Średnia użyta pamięć	845.87	941.92	831.0	684.26
Maksymalna użyta pamięć	885.38	977.68	886.96	735.5
Pamięć JVM (MiB)				
Średnia użyta pamięć JVM	419.49	405.61	414.12	217.06
Maksymalna użyta pamięć JVM	676.47	835.36	684.91	333.27
Wątki (Threads)				
Maksymalna liczba wątków	214.0	1014.0	31.0	37.0
Obciążenie (Load Average)				
Średnie obciążenie	24.09	20.99	15.79	13.88
Maksymalne obciążenie	31.11	27.92	19.51	16.26

5.1.3. Analiza

W przypadku 1000 równoczesnych użytkowników wszystkie testowane rozwiązania osiągnęły 0% błędów. Zauważalna jest jednak istotna różnica w przepustowości oraz czasach odpowiedzi. Aplikacja oparta o WebFlux uzyskała najwyższą liczbę żądań na sekundę oraz relatywnie najniższe opóźnienia zarówno średnie, jak i medianę, P90 oraz P95. Rozwiązanie z wątkami wirtualnymi również wypadło korzystnie, zapewniając drugą co do wielkości przepustowość i stosunkowo niskie czasy odpowiedzi, wyraźnie przewyższając klasyczne implementacje Spring MVC.

Porównując obie aplikacje MVC oparte na klasycznych wątkach: web-mvc-app oraz web-mvc-app-2, należy podkreślić, że web-mvc-app-2 to ten sam kod co web-mvc-app, ale z podniesioną liczbą wątków. Intencją było zwiększenie przepustowości, jednak w praktyce doprowadziło to do gorszej wydajności. Wzrost liczby wątków na stałej liczbie rdzeni procesora skutkuje częstszym przełączaniem kontekstu (ang. context switching), co zwiększa obciążenie CPU. Procesor, zamiast efektywnie przetwarzać żądania, poświęca więcej czasu na zarządzanie wieloma aktywnymi wątkami. Ponadto zwiększenie liczby wątków systemowych powoduje dodatkowe zużycie pamięci. Każdy wątek wymaga pewnego zasobu jak na przykład stosu, co skutkuje wzrostem zużycia pamięci RAM. Efektem jest obniżenie faktycznej przepustowości oraz pogorszenie czasów reakcji w porównaniu do pierwotnych ustawień, w których liczba wątków była niższa.

Dla obciążenia 5000 równoczesnych użytkowników różnice te są jeszcze wyraźniejsze. Klasyczne aplikacje Spring MVC notują rosnący odsetek nieudanych żądań z powodu ograniczonej puli wątków oraz wyraźne wydłużenie czasów odpowiedzi, do kilkudziesięciu sekund w skrajnych przypadkach. WebFlux utrzymuje stabilność i niską zmienność czasów odpowiedzi, notując 0% błędów, a wariant z wątkami wirtualnymi wykazuje znacznie lepszą skalowalność i przewidywalność w porównaniu z tradycyjnymi podejściami.

Z punktu widzenia wykorzystania zasobów CPU i pamięci, WebFlux odznaczał się wysoką wydajnością przetwarzania w przeliczeniu na pojedyncze żądanie, przy stosunkowo wysokim średnim zużyciu CPU. Mimo to uzyskana wyższa przepustowość i krótsze czasy odpowiedzi wskazują na efektywniejsze wykorzystanie dostępnych zasobów. Wirtualne wątki, choć nie osiągnęły tak wysokiej przepustowości jak WebFlux, zapewniały znacząco lepszą skalowalność oraz stabilność czasów odpowiedzi i niższe opóźnienia w porównaniu do klasycznych rozwiązań opartych na standardowych wątkach.

5.1.4. Wnioski

- **Lepsza skalowalność i stabilność podejść asynchronicznych i wykorzystujących wirtualne wątki:** Zarówno WebFlux, jak i web-mvc-vt-app wykazały wyraźnie większą skalowalność, stabilność czasów odpowiedzi oraz wyższą przepustowość niż klasyczne implementacje Spring MVC.
- **Niska liczba nieudanych żądań przy wysokim obciążeniu:** Przy 5000 użytkowników

WebFlux utrzymał 0% błędów oraz przewidywalne, stosunkowo krótkie czasy odpowiedzi, co wskazuje na lepsze radzenie sobie z rosnącą liczbą równoległych żądań.

- **Efektywność wykorzystania zasobów:** WebFlux, mimo wyższego średniego zużycia CPU, zapewnia znacząco wyższą przepustowość i krótsze opóźnienia, co finalnie przekłada się na efektywniejsze wykorzystanie zasobów. Wirtualne wątki pozycjonują się pomiędzy WebFlux a klasycznymi wątkami, zapewniając lepszą skalowalność niż standardowe MVC, bez konieczności nadmiernego zwiększania liczby wątków systemowych.
- **Negatywny efekt nadmiernego zwiększania liczby wątków:** Próba poprawy wydajności przez zwiększenie liczby wątków, jak w przypadku web-mvc-app-2, przyniosła efekt odwrotny do zamierzonego. Przy ograniczonej liczbie rdzeni procesora i rosnącej liczbie wątków występuje wzrost kosztów przełączania kontekstu, co negatywnie wpływa na przepustowość, czasy odpowiedzi i zużycie pamięci.
- **Maksymalne czasy odpowiedzi jako kluczowy czynnik:** Klasyczne Spring MVC (bez wirtualnych wątków) charakteryzowały się długimi, nieprzewidywalnymi opóźnieniami przy dużym obciążeniu. Zarówno WebFlux, jak i podejście z wirtualnymi wątkami, zapewniały bardziej przewidywalne i stabilne czasy obsługi żądań.

5.2. Scenariusz 2: Opóźnienia blokujące

5.2.1. Opis

Drugi scenariusz symuluje opóźnienia wynikające z komunikacji z zewnętrznymi serwisami lub wykonywania operacji blokujących. W implementacji wykorzystano metodę `Thread.sleep`, która wstrzymuje działanie wątku po otrzymaniu zapytania na określony czas, imitując opóźnienie.

Ten scenariusz pozwala na ocenę zachowania aplikacji w sytuacjach, gdy występują opóźnienia niezależne od aplikacji, na przykład podczas oczekiwania na odpowiedź z zewnętrznego API.

5.2.2. Wyniki

5.2.3. Analiza

W scenariuszu z blokującymi opóźnieniami (Thread.sleep) tradycyjne podejście Spring MVC jest najmniej wydajne: przy 1000 użytkowników osiąga bardzo długie czasy odpowiedzi i niską przepustowość, a przy 5000 użytkowników generuje znaczący odsetek nieudanych żądań oraz bardzo wysokie opóźnienia.

Tabela 5.3. Porównanie wydajności API api-delay dla scenariusza 1000 użytkowników

	web-mvc-app	web-mvc-app-2	web-mvc-vt-app	webflux-app
Zadania HTTP				
Nieudane żądania (%)	0.00%	0.00%	0.00%	0.00%
Liczba żądań na sekundę	391.93	1802.84	1802.92	1813.45
Czas żądania (ms)				
Minimalny czas	502.09	500.67	500.77	500.69
Maksymalny czas	5010.0	2600.0	2620.0	646.55
Średni czas	2380.0	509.35	509.47	506.23
Mediana	2500.0	504.36	505.20	504.37
P90	2510.0	509.0	511.71	510.04
P95	2510.0	514.95	517.2	515.73
Użycie CPU (%)				
Średnie użycie CPU	5.09	10.54	10.44	11.15
Maksymalne użycie CPU	7.22	13.09	13.07	11.68
Średnie użycie CPU systemu JVM	5.32	14.09	12.25	13.18
Maksymalne użycie CPU systemu JVM	22.11	26.78	27.64	26.45
Średnie użycie CPU procesu JVM	4.47	13.55	10.07	11.81
Maksymalne użycie CPU procesu JVM	20.17	26.31	26.86	26.68
Pamięć kontenera (MiB)				
Średnia użyta pamięć	463.23	877.65	784.42	529.6
Maksymalna użyta pamięć	486.76	923.96	847.57	544.78
Pamięć JVM (MiB)				
Średnia użyta pamięć JVM	209.28	397.78	430.53	184.74
Maksymalna użyta pamięć JVM	272.56	658.79	748.69	329.84
Wątki (Threads)				
Maksymalna liczba wątków	214.0	1014.0	31.0	49.0
Obciążenie (Load Average)				
Średnie obciążenie	3.36	64.55	14.65	6.04
Maksymalne obciążenie	4.12	191.41	38.57	8.12

Tabela 5.4. Porównanie wydajności API api-delay dla scenariusza 5000 użytkowników

	web-mvc-app	web-mvc-app-2	web-mvc-vt-app	webflux-app
Zadania HTTP				
Nieudane żądania (%)	8.44%	0.91%	0.28%	0.00%
Liczba żądań na sekundę	414.04	1816.3	5954.65	8475.16
Czas żądania (ms)				
Minimalny czas	501.13	501.08	500.77	500.62
Maksymalny czas	60000.0	60000.0	60000.0	1350.0
Średni czas	11210.0	2340.0	709.94	541.34
Mediana	12190.0	2490.0	582.74	528.75
P90	14330.0	2520.0	695.93	589.07
P95	19130.0	2570.0	746.06	611.53
Użycie CPU (%)				
Średnie użycie CPU	3.81	13.13	35.4	36.6
Maksymalne użycie CPU	4.17	14.62	43.38	43.56
Średnie użycie CPU systemu JVM	15.56	14.61	42.96	37.17
Maksymalne użycie CPU systemu JVM	26.32	48.28	81.52	50.67
Średnie użycie CPU procesu JVM	13.4	13.91	40.54	36.34
Maksymalne użycie CPU procesu JVM	22.5	40.92	65.14	45.64
Pamięć kontenera (MiB)				
Średnia użyta pamięć	660.22	951.12	1328.88	890.38
Maksymalna użyta pamięć	736.64	1008.17	1385.1	919.16
Pamięć JVM (MiB)				
Średnia użyta pamięć JVM	166.17	457.22	950.97	434.39
Maksymalna użyta pamięć JVM	202.73	715.92	1239.92	675.57
Wątki (Threads)				
Maksymalna liczba wątków	214.0	1014.0	31.0	49.0
Obciążenie (Load Average)				
Średnie obciążenie	1.55	7.81	9.83	11.77
Maksymalne obciążenie	2.03	9.62	13.82	15.67

Zwiększenie liczby wątków poprawia sytuację, obniżając czasy odpowiedzi i zwiększając przepustowość, lecz pociąga za sobą wysokie zużycie zasobów pamięci i CPU oraz nie osiąga wyników zbliżonych do podejść asynchronicznych.

Aplikacja oparta na wirtualnych wątkach radzi sobie znacznie lepiej. W porównaniu do klasycznych wątków pozwala utrzymać niskie opóźnienia, wysoką przepustowość i mniejsze zużycie zasobów, co wynika z lekkiej natury wirtualnych wątków i ich łatwej skalowalności przy operacjach blokujących.

Najlepszy wynik uzyskuje jednak aplikacja reaktywna. Dzięki asynchronicznej obsłudze I/O i braku blokad na poziomie wątków, nawet blokujące opóźnienia nie degradują znacząco wydajności. WebFlux zachowuje najwyższą przepustowość, najniższe opóźnienia oraz 0% nieudanych żądań przy wysokim obciążeniu.

5.2.4. Wnioski

Różnice wynikają przede wszystkim z podejścia do obsługi blokujących operacji:

- Klasyczny MVC bez zwiększania liczby wątków staje się wąskim gardłem przy opóźnieniach.
- Zwiększenie liczby wątków poprawia wyniki, ale kosztem wysokiego zużycia zasobów i nadal gorszej skalowalności.
- Wirtualne wątki zdecydowanie poprawiają skalowalność i redukują koszt obsługi wielu blokujących zadań, zapewniając lepszą wydajność niż klasyczne podejścia.
- Najlepsze rezultaty osiąga rozwiązanie reaktywne, które dzięki asynchroniczności minimalizuje wpływ blokad na całkowitą przepustowość i czasy odpowiedzi.

Podsumowując, w scenariuszach z blokującymi opóźnieniami asynchroniczne podejścia WebFlux oraz wirtualne wątki zapewniają znacznie lepszą skalowalność i stabilność wydajności niż klasyczne MVC, nawet po zwiększeniu liczby wątków w serwerze aplikacyjnym.

5.3. Scenariusz 3: Zapytania do bazy danych

5.3.1. Opis

Trzeci scenariusz obejmuje obsługę żądań wymagających interakcji z bazą danych PostgreSQL. Zapytanie SQL pobiera wszystkie 70 rekordów z tabeli *products*.

Tabela 5.5. Schemat tabeli `products`

Nazwa kolumny	Typ danych
`product_id`	`smallint NOT NULL`
`product_name`	`character varying(40) NOT NULL`
`supplier_id`	`smallint`
`category_id`	`smallint`
`quantity_per_unit`	`character varying(20)`
`unit_price`	`real`
`units_in_stock`	`smallint`
`units_on_order`	`smallint`
`reorder_level`	`smallint`
`discontinued`	`integer NOT NULL`

5.3.2. Wyniki

Tabela 5.6. Porównanie wydajności API api-db dla scenariusza 1000 użytkowników

	web-mvc-app	web-mvc-app-2	web-mvc-vt-app	webflux-app
Zadania HTTP				
Nieudane żądania (%)	0.00%	0.00%	0.00%	0.00%
Liczba żądań na sekundę	4308.63	4631.38	4376.69	3431.99
Czas żądania (ms)				
Minimalny czas	1.01	950.27	946.38	3.05
Maksymalny czas	2540.0	6750.0	3360.0	954.78
Średni czas	213.47	198.7	210.3	267.6
Mediana	149.07	141.64	142.14	200.83
P90	456.43	481.2	527.87	627.96
P95	576.08	629.98	579.59	687.2
Użycie CPU (%)				
Średnie użycie CPU	41.23	42.35	45.02	63.47
Maksymalne użycie CPU	42.16	42.53	45.37	64.35
Średnie użycie CPU systemu JVM	61.88	62.28	64.09	79.22
Maksymalne użycie CPU systemu JVM	82.19	77.32	87.86	89.84
Średnie użycie CPU procesu JVM	45.55	46.95	49.49	71.15
Maksymalne użycie CPU procesu JVM	79.96	71.93	86.12	90.43
Pamięć kontenera (MiB)				
Średnia użyta pamięć	512.48	871.07	911.27	704.73
Maksymalna użyta pamięć	535.23	972.0	1014.75	731.24
Pamięć JVM (MiB)				
Średnia użyta pamięć JVM	245.9	399.46	490.5	253.44
Maksymalna użyta pamięć JVM	329.83	623.47	876.54	421.51
Wątki (Threads)				
Maksymalna liczba wątków	215.0	1015.0	32.0	38.0
Obciążenie (Load Average)				
Średnie obciążenie	14.14	14.97	13.92	14.79
Maksymalne obciążenie	18.58	18.8	17.64	18.62

Tabela 5.7. Porównanie wydajności API api-db dla scenariusza 5000 użytkowników

	web-mvc-app	web-mvc-app-2	web-mvc-vt-app	webflux-app
Zadania HTTP				
Nieudane żądania (%)	0.56%	0.63%	0.40%	0.00%
Liczba żądań na sekundę	3783.88	3978.75	3164.93	2990.6
Czas żądania (ms)				
Minimalny czas	1.04	1.01	976.62	3.3
Maksymalny czas	60000.0	59990.0	60000.0	6170.0
Średni czas	1120.0	1060.0	1370.0	1570.0
Mediana	634.94	621.23	770.12	1130.0
P90	2430.0	1490.0	3030.0	3700.0
P95	2670.0	2480.0	3410.0	3820.0
Użycie CPU (%)				
Średnie użycie CPU	41.38	37.53	52.86	70.96
Maksymalne użycie CPU	42.02	43.66	56.33	79.01
Średnie użycie CPU systemu JVM	62.02	62.51	64.97	81.33
Maksymalne użycie CPU systemu JVM	79.01	77.08	91.88	91.81
Średnie użycie CPU procesu JVM	45.66	47.35	53.1	74.94
Maksymalne użycie CPU procesu JVM	74.27	75.64	92.93	91.28
Pamięć kontenera (MiB)				
Średnia użyta pamięć	795.14	969.58	1364.12	911.23
Maksymalna użyta pamięć	892.78	1025.06	1413.02	939.15
Pamięć JVM (MiB)				
Średnia użyta pamięć JVM	391.45	459.98	973.16	416.81
Maksymalna użyta pamięć JVM	643.94	674.99	1290.39	609.1
Wątki (Threads)				
Maksymalna liczba wątków	215.0	1015.0	32.0	38.0
Obciążenie (Load Average)				
Średnie obciążenie	18.17	18.65	16.73	14.6
Maksymalne obciążenie	28.43	28.38	23.36	18.1

5.3.3. Analiza

Przy 1000 równoczesnych użytkowników, wszystkie aplikacje obsłużyły żądania bez błędów. Warto jednak zwrócić uwagę na różnice w przepustowości oraz czasach odpowiedzi. Podejście klasyczne z większą liczbą wątków zapewniło najwyższą przepustowość oraz bardzo dobre czasy odpowiedzi. Aplikacja oparta o wirtualne wątki osiągała zbliżone wyniki do *web-mvc-app-2*, lecz nieznacznie gorsze czasy. Natomiast aplikacja reaktywna, mimo nieco niższej przepustowości i wyższych czasów odpowiedzi, charakteryzowała się niższym użyciem pamięci JVM, a także najmniejszym czasem maksymalnej odpowiedzi. Klasyczna aplikacja *web-mvc-app* plasowała się pomiędzy tymi podejściami, osiągając umiarkowaną przepustowość oraz czasy odpowiedzi.

Przy obciążeniu 5000 użytkowników sytuacja uległa zmianie. Wszystkie klasyczne podejścia zanotowały wzrost nieudanych żądań, co świadczy o trudniejszym skalowaniu się pod większym obciążeniem. Aplikacja webflux-app utrzymała 0% błędów, co świadczy o stabilności reaktywnego podejścia przy rosnącej liczbie równoczesnych żądań. Osiągała ona jednak niższą przepustowość niż klasyczne podejścia, a czasy odpowiedzi wskazują na trudności w utrzymaniu niskich opóźnień przy tak dużym obciążeniu.

Warto zwrócić uwagę na wykorzystanie zasobów. Podczas wysokiego obciążenia aplikacja webflux-app charakteryzowała się wyższym zużyciem CPU oraz niskim zużyciem pamięci. Rozwiązanie z wirtualnymi wątkami, mimo lepszej skalowalności przy

mniejszych obciążeniach, przy 5000 użytkowników nie uzyskało przewagi ani pod względem przepustowości, ani czasów, a także zanotowało błędy (0.4%).

5.3.4. Wnioski

- Przy niższym obciążeniu najlepszą przepustowość i czasy uzyskało klasyczne podejście z dużą liczbą wątków, co sugeruje, że zwiększona liczba wątków systemowych może pomóc, o ile obciążenie nie jest wysokie.
- Wirtualne wątki pozwalają osiągnąć wyniki zbliżone do web-mvc-app-2 przy mniejszej liczbie wątków i umiarkowanym koszcie zasobów, jednak nie zapewniły one znaczącej przewagi przy 5000 użytkowników.
- Podejście reaktywne odznacza się wysoką stabilnością, lecz odbywa się to kosztem wyższych opóźnień i niższej przepustowości. Sugeruje to, że w warunkach bardzo dużego obciążenia reakcja na blokujące wywołania do bazy danych staje się wąskim gardłem, mimo asynchroniczności.
- Klasyczne podejścia przy bardzo dużym obciążeniu tracą na stabilności, mimo często niższych opóźnień niż w przypadku webflux-app.

Podsumowując, podczas obsługi żądań wymagających dostępu do bazy danych wybór podejścia zależy od priorytetów: jeśli kluczowa jest stabilność i unikanie błędów przy bardzo wysokim obciążeniu, podejście reaktywne będzie odpowiednie, mimo wyższych opóźnień. Jeśli zaś ważniejsza jest maksymalna przepustowość przy umiarkowanym obciążeniu i akceptowalnych czasach, to zwiększenie liczby wątków lub użycie wirtualnych wątków może okazać się korzystne.

5.3.5. Problem wątków wirtualnych

Wyniki testów pokazują, że w przypadku zastosowania wirtualnych wątków wydajność aplikacji była gorsza niż w przypadku korzystania z klasycznych wątków, co jest sprzeczne z oczekiwaniami i naturą wątków wirtualnych.

Pula połączeń Powodem gorszych rezultatów w analizowanym przypadku jest czas oczekiwania na uzyskanie połączenia do bazy danych z puli bazodanowej (HikariCP). Przy domyślnych ustawieniach puli (10 połączeń) i dużej liczbie równoległych żądań, wątki wirtualne często musiały oczekiwać na zwolnienie połączenia. W przypadku wirtualnych wątków, których może być znacznie więcej niż wątków systemowych, czas oczekiwania na wolne połączenie może się znacząco wydłużyć. Niewystarczająca pula połączeń staje się wąskim gardłem oraz wpływa negatywnie na opóźnienia i przepustowość.

Zgodnie z dokumentacją HikariCP[32], rozmiar puli powinien być mały, z przewagą wątków oczekujących na połączenia. Optymalna pula jest ustawiona na granicy liczby zapytań, które baza danych może przetwarzać jednocześnie – zwykle nieco powyżej core_count × 2. Rozmiar puli połączeń można oszacować za pomocą wzoru:

$$\text{Connections} = (\text{core\_count} \times 2) + \text{effective\_spindle\_count} \tag{2}$$

gdzie:

- **core_count** – liczba rdzeni CPU,
- **effective_spindle_count** – liczba aktywnych dysków bazy danych.

Przy dużej liczbie wirtualnych wątków, dziesięć połączeń w puli to zbyt mało, powodując nadmierne wykorzystanie kolejki oczekujących na połączenie. Efektem jest dłuższy czas uzyskania połączenia, który dla konfiguracji przed optymalizacją wynosił nawet ok. 161 ms w testach dla 1000 użytkowników, co znacząco ograniczało korzyści płynące z użycia wirtualnych wątków.

Wprowadzenie semafora o pojemności 100, stosowanego przy wywołaniu metod repozytorium w serwisie, ograniczyło liczbę jednoczesnych zapytań do bazy i tym samym zmniejszyło obciążenie puli połączeń. To podejście skutecznie skróciło średni czas uzyskania połączenia do ok. 19.3 ms, co przełożyło się na poprawę średnich czasów odpowiedzi i przepustowości, zwłaszcza dla 1000 użytkowników. Jednak przy 5000 użytkowników poprawa wciąż jest niewielka.

Tabela 5.8. Porównanie Connection Acquire Time dla różnych wariantów i wprowadzenia semafora (1000 użytkowników)

Wariant	Connection Acquire Time (ms)
web-mvc-app	40.8
web-mvc-app-2	167.0
web-mvc-vt (przed semaforem)	161.0
web-mvc-vt (po semaforze)	19.3

Listing 2. Semafor w metodzie zapytania do bazy danych

```
private final Semaphore dbSemaphore = new Semaphore(100);

    public List<Product> queryDatabase() {
        try {
            dbSemaphore.acquire();
            return productRepository.findAll();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } finally {
            dbSemaphore.release();
        }
    }
```

Thread Local Następnym aspektem jest wewnętrzne użycie ThreadLocal przez bibliotekę HikariCP.

W Apache Tomcat zarządzanie wątkami odbywa się za pomocą puli wątków. Podczas korzystania z klasycznych wątków systemowych, gdy serwer otrzymuje nowe żądanie, nie tworzy dla niego nowego wątku od podstaw. Zamiast tego korzysta z wcześniej utworzonych wątków dostępnych w puli. Jeśli wszystkie wątki w puli są zajęte, nowe żądania są umieszczane w kolejce i czekają na zwolnienie któregoś z wątków. W przypadku wątków wirtualnych, dla każdego zapytania tworzony jest nowy wątek. Omówione to zostało w rozdziale 2. Wewnętrzna implementacja HikariCP działa tak, że gdy wątek wywołuje metodę `getConnection()`, aby uzyskać połączenie, wywołuje to metodę `ConcurrentBag.borrow()`, która najpierw podejmie próbę uzyskania nieużywanego połączenia z listy ostatnio używanych połączeń wątku. W przypadku wątków wirtualnych lista ta nigdy nie będzie zawierać ostatnio zamkniętych połączeń, więc wątek będzie pobierał je z głównej listy połączeń. Gdy wątki wirtualne zostaną zakończone, przechowywane w nich połączenia do bazy danych nie mogą być ponownie wykorzystane. Sprawia to, że korzystanie z wątków wirtualnych stanowi dodatkowe obciążenie, ponieważ połączenie nie może być współdzielone pomiędzy różnymi zapytaniami wykonywanymi przez inne wątki. Jest to mniej wydajne, ale nadal poprawne.

Widoczne jest to dla aplikacji web-mvc-app-2 i web-mvc-vt-app dla 1000 użytkowników, gdzie czas oczekiwania na połączenie jest podobny, ale przepustowość aplikacji bez wątków wirtualnych jest większa, ponieważ częstotliwość tworzenia nowych połączeń jest mniejsza.

Tabela 5.9. Porównanie wydajności web-mvc-vt przed i po wprowadzeniu semafora dla 1000 i 5000 użytkowników

	1000 użytkowników		5000 użytkowników	
	web-mvc-vt	web-mvc-vt (semafor)	web-mvc-vt	web-mvc-vt (semafor)
Nieudane żądania (%)	0.00%	0.00%	0.40%	0.36%
Liczba żądań/s	4376.69	5224.46	3164.93	3538.29
Średni czas (ms)	210.3	175.83	1370.0	1200.0
Mediana (ms)	142.14	156.63	770.12	843.54
P90 (ms)	527.87	231.58	3030.0	2250.0
P95 (ms)	579.59	577.35	3410.0	3610.0
Średnie użycie CPU (%)	45.02	46.43	52.86	49.26
Średnia pamięć kontenera (MiB)	911.27	828.66	1364.12	1339.1
Średnia pamięć JVM (MiB)	490.5	447.06	973.16	952.28
Maks. liczba wątków	32.0	32.0	32.0	32.0
Średnie obciążenie (Load Avg)	13.92	13.22	16.73	13.19

Z analizy wynika, że:

- Wirtualne wątki mają potencjał do obsługi dużej liczby równoległych żądań, lecz muszą być wspierane odpowiednio skonfigurowaną pulą połączeń lub mechanizmami kontrolującymi obciążenie.
- Niewystarczająca pula połączeń do bazy danych przy dużej liczbie wątków staje się wąskim gardłem, znacznie podnosząc czasy oczekiwania i pogarszając wydajność.
- Zastosowanie semafora lub innych form ograniczania współbieżnych zapytań do bazy może istotnie skrócić czas oczekiwania na połączenie, co przekłada się na

wzrost przepustowości i poprawę czasów odpowiedzi, szczególnie przy umiarkowanym obciążeniu.

- Przy bardzo wysokim obciążeniu konieczne może być dalsze skalowanie zasobów (zwiększenie rozmiaru puli połączeń, optymalizacja bazy danych).
- Należy zwracać uwagę na zachowanie bibliotek oraz ich kompatybilność z wątkami wirtualnymi.

5.4. Scenariusz 4: Operacje na plikach

5.4.1. Opis

Czwarty scenariusz testuje wydajność aplikacji podczas wykonywania operacji na systemie plików. Aplikacja realizuje następujące kroki:

1. Utworzenie katalogu roboczego, jeśli nie istniał.
2. Wygenerowanie unikalnej nazwy pliku.
3. Zapis pliku o rorzmiarze 4kb na dysku.
4. Odczyt zapisanego pliku.
5. Usunięcie pliku po zakończeniu operacji.

W przypadku aplikacji opartej na Spring WebFlux, wszystkie powyższe operacje były realizowane w sposób asynchroniczny i nieblokujący, z wykorzystaniem reaktywnych strumieni danych. Natomiast w aplikacji Spring MVC operacje te były wykonywane synchronicznie.

5.4.2. Wyniki

Tabela 5.10. Porównanie wydajności API api-file dla 1000 użytkowników

	web-mvc-app	web-mvc-app-2	web-mvc-vt-app	webflux-app
Zadania HTTP				
Nieudane żądania (%)	0.00%	0.00%	0.00%	0.00%
Liczba żądań/s	6281.22	5526.34	6485.63	6598.39
Czas żądania (ms)				
Minimalny czas	0.57	0.62	0.54	0.67
Maksymalny czas	2350.0	4080.0	2480.0	1180.0
Średni czas	146.02	166.36	141.41	138.9
Mediana	110.59	6.35	93.17	125.61
P90	241.38	470.99	279.29	209.18
P95	375.24	540.92	345.66	266.97
Użycie CPU (%)				
Średnie użycie CPU	41.29	40.56	41.01	47.16
Maksymalne użycie CPU	42.8	41.49	41.81	48.26
Średnie użycie CPU systemu JVM	50.06	48.62	50.4	53.9
Maksymalne użycie CPU systemu JVM	69.73	73.59	73.65	81.76
Średnie użycie CPU procesu JVM	45.07	44.89	44.25	52.4
Maksymalne użycie CPU procesu JVM	69.46	75.62	76.08	86.53
Pamięć kontenera (MiB)				
Średnia użyta pamięć	975.75	1265.3	1048.48	953.62
Maksymalna użyta pamięć	1449.43	1767.89	1533.18	1444.7
Pamięć JVM (MiB)				
Średnia użyta pamięć JVM	222.43	403.4	229.5	145.52
Maksymalna użyta pamięć JVM	291.5	706.05	305.28	180.98
Wątki (Threads)				
Maksymalna liczba wątków	214.0	1014.0	275.0	116.0
Obciążenie (Load Average)				
Średnie obciążenie	139.54	625.56	177.2	23.03
Maksymalne obciążenie	194.21	910.97	247.5	29.1

Tabela 5.11. Porównanie wydajności API api-file dla 5000 użytkowników

	web-mvc-app	web-mvc-app-2	web-mvc-vt-app	webflux-app
Zadania HTTP				
Nieudane żądania (%)	0.24%	0.46%	0.20%	0.00%
Liczba żądań/s	5507.36	4545.14	5170.91	7613.39
Czas żądania (ms)				
Minimalny czas	0.52	0.59	0.57	0.73
Maksymalny czas	60000.0	60000.0	60000.0	4340.0
Średni czas	761.93	948.91	827.07	606.5
Mediana	513.07	528.64	507.21	495.06
P90	1080.0	1310.0	1940.0	927.37
P95	2150.0	2000.0	2250.0	1760.0
Użycie CPU (%)				
Średnie użycie CPU	40.46	21.86	39.38	49.43
Maksymalne użycie CPU	43.8	41.31	42.41	55.48
Średnie użycie CPU systemu JVM	47.37	49.55	51.33	57.5
Maksymalne użycie CPU systemu JVM	68.59	69.73	74.03	86.13
Średnie użycie CPU procesu JVM	40.16	44.78	44.08	55.5
Maksymalne użycie CPU procesu JVM	66.49	73.75	74.39	90.37
Pamięć kontenera (MiB)				
Średnia użyta pamięć	1209.89	1362.75	1297.6	1083.54
Maksymalna użyta pamięć	1693.11	1737.68	1734.62	1687.42
Pamięć JVM (MiB)				
Średnia użyta pamięć JVM	367.99	462.0	382.52	162.79
Maksymalna użyta pamięć JVM	615.27	687.41	614.11	232.46
Wątki (Threads)				
Maksymalna liczba wątków	214.0	1014.0	275.0	128.0
Obciążenie (Load Average)				
Średnie obciążenie	161.62	694.01	190.04	25.85
Maksymalne obciążenie	199.36	951.08	248.31	31.36

5.4.3. Analiza

W scenariuszu 4 różnice pomiędzy podejściem synchronicznym a asynchronicznym stają się wyraźne. Zestawione dane wskazują, że przy 1000 równoczesnych użytkowników aplikacja oparta na WebFlux osiągała najwyższą przepustowość, a także niższe czasy odpowiedzi w porównaniu do rozwiązań MVC.

W przypadku aplikacji Spring MVC z domyślną liczbą wątków osiągnięto dobrą przepustowość i czasy odpowiedzi, co pokazuje, że operacje na plikach nie stanowią tak poważnego wąskiego gardła, jak w przypadku intensywnego korzystania z baz danych czy blokujących zewnętrznych usług. Natomiast wariant *web-mvc-app-2*, z większą liczbą wątków, nie poprawił wydajności w sposób zdecydowany, a nawet osiągnął gorszą medianę czasu niż podstawowy web-mvc-app, co jest analogiczne do obserwacji w innych scenariuszach, gdzie nadmierne zwiększanie liczby wątków prowadzi do większej liczby przełączeń kontekstu.

Wariant z wirtualnymi wątkami przy 1000 użytkowników osiągał bardzo dobrą przepustowość i niskie średnie czasy odpowiedzi. Wskazuje to, że przy umiarkowanym obciążeniu i blokujących operacjach I/O wirtualne wątki mogą poprawnie wykorzystywać zasoby. Parametr, który zwraca uwagę, to maksymalna liczba wątków, która jest większa niż w pozostałych scenariuszach. Powodem tego jest przypinanie wątków (`ang. thread pinning`) podczas wykonywania operacji systemowych przez natywne metody operacji na plikach w Javie, czego dotyczył rozdział 2.5.

Podsumowanie W przypadku operacji blokujących, nadmierny przyrost liczby wątków fizycznych prowadzi do zwiększonej liczby przełączania kontekstu i w efekcie może pogorszyć wydajność. Podobnie jak w scenariuszu "Hello World"czy przy bazie danych, zwiększanie liczby wątków ponad pewną granicę skutkuje pogorszeniem wyników. Dla operacji na plikach z użyciem wirtualnych wątków, gdy obciążenie wzrasta do 5000 użytkowników, zauważalna jest tendencja do wzrostu czasów odpowiedzi i pojawienia się błędów w klasycznych podejściach. W przypadku aplikacji z wątkami wirtualnymi powodem pogorszenia wyników jest ForkJoinPool, który nie nadaje się do zadań, które obejmują blokowanie operacji, ponieważ zwiększa złożoność zarządzania wątkami. WebFlux, wciąż dobrze skaluje się przy dużej liczbie równoczesnych użytkowników, zapewniając 0% błędów i najwyższą przepustowość dzięki asynchronicznej naturze operacji I/O.

5.4.4. Wnioski

- **Asynchroniczność i nieblokujące operacje I/O wygrywają** WebFlux ponownie okazuje się skuteczny w obsłudze dużej liczby równoczesnych żądań wymagających operacji na plikach, dzięki asynchronicznej realizacji zadań I/O. Przekłada się to na wysoką przepustowość, stabilność oraz utrzymanie niskich czasów odpowiedzi nawet przy 5000 użytkowników.
- **Wirtualne wątki są nieefektywne** Wirtualne wątki zarządzane przez `ForkJoinPool` nie są optymalne dla natywnych systemowych zadań. Gdy obciążenie rośnie, pojawia się efekt przypinania wątków i zwiększania liczby wątków systemowych, co prowadzi do spadku wydajności. Przy 1000 użytkowników web-mvc-vt-app radzi sobie radzi sobie lepiej niż aplikacja bez wątków wirtualnych, ponieważ jej maksymalna pula wątków (256) ForkJoinPool jest wieksza od domyślnej puli Tomcata (200).
- **Nadmierne zwiększanie liczby wątków w klasycznych podejściach jest nieefektywne** Podobnie jak w poprzednich scenariuszach, zwiększenie liczby wątków nie przynosi korzyści. Nadmiar wątków przy 5000 użytkowników skutkuje jeszcze gorszymi opóźnieniami i znikomym zyskiem w przepustowości.

5.5. WebFlux z wykorzystaniem wątków wirtualnych

W aplikacjach reaktywnych zarządzanie zasobami odbywa się na poziomie pętli zdarzeń i puli wątków, które są zoptymalizowane do obsługi dużej liczby równoczesnych zadań bez konieczności tworzenia wielu wątków. Wirtualne wątki mogą być nadmiarowe w takim kontekście, gdyż nie oferują dodatkowych korzyści w porównaniu do już efektywnych mechanizmów reaktywnych. Integracja wirtualnych wątków z istniejącymi bibliotekami i frameworkami reaktywnymi może wprowadzać dodatkową złożoność bez realnych korzyści. Wyjątkiem jest obsługa zadań blokujących w ramach przepływu reaktywnego, które są wykonywane na oddzielnych wątkach niż pętla zdarzeń.

5.5.1. Nowości w Reactor Core

W Reactor Core 3.6.0 wprowadzono zmiany, które wspierają wykorzystanie wątków wirtualnych w aplikacjach opartych na WebFlux. Jedną z nowości są optymalizacje wydajności, które wprowadzają ulepszone algorytmy harmonogramowania, lepiej dostosowane do charakterystyki wątków wirtualnych. Gdy mamy do czynienia z blokującymi wywołaniami w Reactorze, używamy następującej konstrukcji:

```
Mono.fromCallable(() -> {
    return blockingOperation();
}).subscribeOn(Schedulers.boundedElastic());
```

W metodzie `subscribeOn()` podawany jest `Scheduler`, który tworzy dedykowany wątek do wykonania tej blokującej operacji. Oznacza to jednak, że wątek systemowy zostanie ostatecznie zablokowany.

Od wersji Java 21+ oraz `reactor-core` 3.6.x można zastąpić domyślną implementację nową `BoundedElasticThreadPerTaskScheduler`, aby używać wątków wirtualnych zamiast wątków platformowych w `Schedulers.boundedElastic()`.

Aby to zrobić, należy ustawić zmienną systemową:

```
-Dreactor.schedulers.defaultBoundedElasticOnVirtualThreads=true
```

Dzięki temu `Schedulers.boundedElastic()` będzie wykorzystywał wątki wirtualne.

5.5.2. Problemy z Netty

Netty to domyślny serwer używany przez WebFlux. Tradycyjnie operuje na modelu wątków opartych na puli. Netty został pierwotnie zaprojektowany z myślą o tradycyjnych wątkach, co powoduje, że niektóre jego wymagają modyfikacji lub dodatkowych mechanizmów synchronizacji, aby efektywnie współpracować z wątkami wirtualnymi. Na czas pisania pracy serwer ten nie wspiera wątków wirtualnych.

6. Podsumowanie

6.1. Podsumowanie i wnioski

Przeprowadzone eksperymenty i analizy przedstawione w niniejszej pracy pozwalają na porównanie dwóch podejść do obsługi współbieżności i asynchroniczności w aplikacjach webowych: reaktywnego programowania z wykorzystaniem Spring WebFlux oraz stosunkowo nowej technologii wirtualnych wątków w Javie z wykorzystaniem klasycznego imperatywnego podejścia. Wyniki testów obejmujących różne scenariusze obciążenia od prostych operacji "Hello World", poprzez symulowane opóźnienia, interakcje z bazą danych, aż po operacje na systemie plików uwidaczniają zróżnicowane zachowanie, wady i zalety każdego z tych podejść.

Reaktywne programowanie zapewnia wysoką skalowalność i stabilność przy bardzo dużej liczbie równoczesnych żądań, szczególnie w scenariuszach z blokującymi operacjami I/O, takimi jak dostęp do zewnętrznych API czy bazy danych. Architektura oparta na przepływach reaktywnych umożliwia nieblokujące przetwarzanie, skutkując wysoką przepustowością i niskimi opóźnieniami nawet przy rosnącym obciążeniu. Dodatkową zaletą jest względna przewidywalność czasów odpowiedzi i brak wyraźnych skoków opóźnień. Jednak podejście reaktywne wymaga zmiany paradygmatu myślenia, kod staje się bardziej złożony w utrzymaniu, a posługiwanie się funkcjami zwrotnymi (ang. callback), strumieniami danych i nieblokującymi API stanowi wyzwanie w porównaniu do tradycyjnego, imperatywnego stylu programowania. Reaktywność wymusza również bardziej skomplikowaną integrację z ekosystemem reaktywnym, biblioteki i narzędzia muszą być kompatybilne z asynchronicznym i nieblokującym modelem.

Wirtualne wątki natomiast oferują prostszą semantykę z punktu widzenia dewelopera. Dzięki nim można pisać kod w stylu imperatywnym, bez konieczności zmiany paradygmatu na reaktywny, a zarazem skorzystać z potencjału asynchroniczności i równoległości. W wielu przypadkach wirtualne wątki mogą osiągnąć wydajność zbliżoną do rozwiązań reaktywnych, unikając zawiłości kodu reaktywnego. Jednakże technologia wirtualnych wątków jest stosunkowo nowa i wciąż dojrzewa. Wymaga ona uważności przy integracji z niektórymi bibliotekami czy narzędziami, co pokazał na przykład problem z pulą połączeń do bazy danych HikariCP w scenariuszu 3, czy wywołaniami systemowymi i operacjami na plikach w scenariuszu 4. Dodatkowo, w sytuacjach natywnych systemowych operacji, dochodzi do przypinania wirtualnych wątków do fizycznych, co prowadzi do niekontrolowanego wzrostu liczby wątków i zwiększonego przełączania kontekstu. W efekcie wydajność może ulec pogorszeniu.

Wnioski końcowe:

- **Reaktywne programowanie (WebFlux):** Zapewnia wysoką skalowalność, stabilność i przewidywalność wydajności przy bardzo dużym obciążeniu I/O. Jest szczególnie efektywne w projektach wymagających obsługi wielu równoczesnych, nieblokujących operacji. Wadą jest wyższa złożoność wynikająca ze zmiany paradygmatu.
- **Wirtualne wątki:** Pozwalają pisać kod w prostszy, imperatywny sposób, osiągając nieraz podobną skalowalność jak podejście reaktywne, bez konieczności stosowania funkcyjnych wzorców reaktywnych. Są jednak technologią młodą, wymagającą uwagi w kwestii kompatybilności z bibliotekami i narzędziami. Przy nieodpowiedniej konfiguracji lub niepoprawnych wykorzystaniu mogą generować nadmierne obciążenie zasobów (m.in. zwiększoną liczbę wątków systemowych).
- **Dobór technologii:** Wybór pomiędzy programowaniem reaktywnym a wirtualnymi wątkami zależy od konkretnych wymagań projektu. Wyniki testów wykazały, że efektywna obsługa żądań przy dużym obciążeniu zależy nie tylko od modelu programowania, ale także od dostępu do zasobów takich jak połączenia do bazy danych czy kompatybilność wykorzystywanych funkcjnalności lub narzędzi z danym modelem. Reaktywny model sprawdzi się tam, gdzie kluczowa jest maksymalna skalowalność i bezbłędna obsługa dziesiątek czy setek tysięcy równoczesnych połączeń. Wirtualne wątki warto rozważyć w przypadkach, gdy istotna jest prostota implementacji i możliwość wykorzystania istniejącego kodu oraz wzorców imperatywnych, przy jednoczesnej konieczności obsługi sporej liczby żądań. W takich sytuacjach właściwa konfiguracja puli wątków, puli połączeń czy ograniczenie blokujących operacji oraz odpowiedni dobór narzedzi pozwoli w pełni wykorzystać potencjał wirtualnych wątków.

Ostatecznie, dobór technologii jest kwestią kompromisu pomiędzy skalowalnością, złożonością implementacji, wydajnością, stabilnością stosowanych zewnętrznych bibliotek oraz preferencjami i kompetencjami zespołu deweloperskiego.

6.2. Dalszy rozwój

W obliczu dynamicznego postępu technologicznego oraz coraz bardziej złożonych wymagań stawianych systemom informatycznym, konieczne jest nieustanne poszukiwanie i rozwijanie nowych metod oraz narzędzi programistycznych. Porównanie ich z tradycyjnym Spring Bootem oraz z wykorzystaniem wirtualnych wątków pozwoli na ocenę ich efektywności w różnych scenariuszach.

W celu pogłębienia analizy i uzyskania bardziej wszechstronnych wyników, warto rozszerzyć badania na inne frameworki takie jak Quarkus czy Micronaut. Również zmiana środowiska testowego na inne, na przykład na platformę ARM, może dostarczyć dodatkowych informacji na temat zachowania aplikacji w różnych warunkach

sprzętowych. Ponadto testy można rozszerzyć o dodatkowe scenariusze lub integracje z innymi systemami, takimi jak Apache Kafka, RabbitMQ czy MongoDB.

Kontynuacja badań w tych obszarach przyczyni się do głębszego zrozumienia zalet i ograniczeń poszczególnych podejść do współbieżności. Pozwoli to na lepsze dostosowanie narzędzi i metod do specyficznych potrzeb projektów, co w efekcie może prowadzić do tworzenia bardziej wydajnych, skalowalnych i łatwiejszych w utrzymaniu systemów informatycznych.

Bibliografia

[1] D. Beronić, P. Pufek, B. Mihaljević i A. Radovan, "On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVM", w *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, 2021. DOI: 10.23919/MIPRO52101.2021.9596855.

[2] J. Bonér, D. Farley, R. Kuhn i M. Thompson, *The Reactive Manifesto v2.0*, https://www.reactivemanifesto.org/, 2014.

[3] B. Brazil, *Prometheus: Up & Running*. O'Reilly Media, Inc., 2018.

[4] A. Catrina, *Performance Evaluation of Virtual Threads in Java 19*, Metropolia University of Applied Sciences (Finland), https://www.theseus.fi/bitstream/handle/10024/812448/Catrina_Alexandru.pdf?sequence=2, 2023.

[5] P. Dakowitz, "Comparing Reactive and Conventional Programming of Java Based Microservices in Containerized Environments", https://reposit.haw-hamburg.de/bitstream/20.500.12738/8321/1/thesis.pdf, prac. mag., HAW Hamburg, 2018.

[6] H. K. Dhalla, "Benchmarking the Performance of RESTful Applications Implemented in Spring Boot Java and MS. Net Core", *Journal of Computing Sciences in Colleges*, t. 36, nr. 3, 2020. adr.: https://dl.acm.org/doi/abs/10.5555/3447080.3447113.

[7] *Docker Documentation*, https://docs.docker.com/, 2024.

[8] *Capturing JVM- and Application-level Metrics. So You Know What's Going On*, https://github.com/dropwizard/metrics, grud. 2017.

[9] N. Dragoni, S. Giallorenzo, A. Lluch Lafuente i in., "Microservices: Yesterday, Today, and Tomorrow", w *Present and Ulterior Software Engineering*, M. Mazzara i B. Meyer, red., Cham: Springer International Publishing, 2017. DOI: 10.1007/978-3-319-67425-4_12.

[10] *Grafana - the Open Platform for Analytics and Monitoring*, https://grafana.com/, 2018.

[11] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes i D. Lea, *Java Concurrency in Practice*. Boston, MA, USA: Addison-Wesley, 2006.

[12] B. Gregg, *Systems Performance, 2nd Edition*. Pearson, 2021.

[13] S. Iwanowski i G. Kozieł, "Comparative Analysis of Reactive and Imperative Approach in Java Web Application Development", *Journal of Computer Sciences Institute (JCSI)*, t. 24, 2022. adr.: https://ph.pollub.pl/index.php/jcsi/article/view/2999/2710.

[14] Z. M. Jiang i A. E. Hassan, "A Survey on Load Testing of Large-Scale Software Systems", *IEEE Transactions on Software Engineering*, t. 41, nr. 11, list. 2015. DOI: 10.1109/TSE.2015.2445340.

[15] L. Urban, *Performance Testing with k6: A Developer's Guide*. Packt Publishing, 2023.

[16] *k6 Documentation*, https://k6.io/docs/, 2024.

[17] OpenJDK Project Loom, *JEP 425: Virtual Threads*, https://openjdk.org/jeps/425, 2022.

[18] K. Mochniej i M. Badurowicz, "Performance Comparison of Microservices Written Using Reactive and Imperative Approaches", *Journal of Computer Sciences Institute (JCSI)*, t. 28, 2023. adr.: https://ph.pollub.pl/index.php/jcsi/article/view/3698/4172.

[19] J. Nickoloff i S. Kuenzli, *Docker in Action, 2nd Edition*. Manning Publications, 2019.

[20] J. Olofsson, *Comparing Virtual Threads and Reactive WebFlux in Spring*, KTH Royal Institute of Technology, Sweden, https://www.diva-portal.org/smash/get/diva2:1763111/FULLTEXT01.pdf, 2023.

[21] A. Navarro, J. Ponge, F. Le Mouël i C. Escoffier, "Considerations for Integrating Virtual Threads in a Java Framework", *International Journal of Web Engineering and Technology*, 2023. adr.: https://inria.hal.science/hal-04112339/document.

[22] A. Prokopec, A. Rosà, D. Leopoldseder i in., "Renaissance: Benchmarking Suite for Parallel Applications on the JVM", w *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*, New York, NY, USA: Association for Computing Machinery, 2019. DOI: 10.1145/3314221.3314637.

[23] *The Reactive Manifesto*, https://www.reactivemanifesto.org/, 2018.

[24] C. Richardson, *Microservices Architecture Pattern*, http://microservices.io/patterns/microservices.html, 2015.

[25] *spring.io*, https://spring.io/, 2017.

[26] *Spring WebFlux Documentation*, https://docs.spring.io/spring-framework/docs/current/reference/web-reactive.html, 2024.

[27] *Spring WebMVC Documentation*, https://docs.spring.io/spring-framework/docs/current/reference/web.html, 2024.

[28] G. Tene, *How Not to Measure Latency*, 2013.

[29] C. Walls, *Spring in Action, 5th Edition*. Manning Publications, 2018.

[30] M. Heckler, *Reactive Spring*. O'Reilly Media, 2021.

[31] J. D. C. Little, "A proof for the queuing formula $L = \lambda W$", *Operations Research*, t. 2, nr. 4, s. 383–387, 1954.

[32] Leo Bayer, *About Pool Sizing*, https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing, 2021.

Wykaz symboli i skrótów

EiTI – Wydział Elektroniki i Technik Informacyjnych
PW – Politechnika Warszawska
API – Interfejs Programistyczny Aplikacji
HTTP – Protokół Transferu Hipertekstu
JVM – Wirtualna Maszyna Java
RPC – Zdalne Wywołanie Procedur
TCP – Protokół Kontroli Transmisji
CPU – Centralna Jednostka Przetwarzająca
RAM – Pamięć o Dostępie Swobodnym
IoT – Internet Rzeczy
I/O – Wejście/Wyjście
RPS – Żądania na Sekundę
MVC – Model-Widok-Kontroler
TSDB – Baza Danych Szeregu Czasowego
JSON – JavaScript Object Notation
VU – Użytkownik Wirtualny (w kontekście k6)
SQL – Język Zapytaniowy SQL
JNI – Java Native Interface

Spis rysunków

Spis tabel

Spis załączników

Załącznik 1. Repozytorium projektu

https://github.com/azexs/vt-webflux-mvc-comparison

Załącznik 2. Kod podejscia imperatywnego

```
1
2  @RestController
3  public class TestController {
4
5      private final TestService testService;
6
7      public TestController(TestService testService) {
8          this.testService = testService;
9      }
10
11     @GetMapping("/api/hello")
12     public String hello() {
13         return "Hello World!";
14     }
15
16     @GetMapping("/api/db")
17     public String getData() {
18         testService.queryDatabase();
19         return "OK";
20     }
21
22     @GetMapping("/api/file")
23     public String getFile() throws IOException {
24         testService.performIOOperation(Paths.get("/tmp"));
25         return "OK";
26     }
27
28     @GetMapping("/api/delay")
29     public String getDelay(@RequestParam(
30             name = "delay",
31             required = false,
32             defaultValue = "500") long delay) throws InterruptedException {
33         Thread.sleep(delay);
34         return "OK";
35     }
36
37 @Service
38 public class TestService {
39
40     private static final int BUFFER_SIZE = 4096;
41     private static final int TOTAL_SIZE = 4096;
42
43     private final ProductRepository productRepository;
44
45     public TestService(ProductRepository productRepository) {
46         this.productRepository = productRepository;
47     }
48
49     public List<Product> queryDatabase() {
50         return productRepository.findAll();
51     }
52
53     public void performIOOperation(Path directory) throws IOException {
```

```
54          Files.createDirectories(directory);
55          String fileName = "testfile_" + UUID.randomUUID() + ".dat";
56          Path filePath = directory.resolve(fileName);
57          writeLargeFile(filePath);
58          readLargeFile(filePath);
59          Files.deleteIfExists(filePath);
60      }
61
62      private void writeLargeFile(Path filePath) throws IOException {
63          Random random = new Random();
64          byte[] buffer = new byte[BUFFER_SIZE];
65          try (SeekableByteChannel channel = Files.newByteChannel(
66              filePath,
67              StandardOpenOption.CREATE,
68              StandardOpenOption.WRITE)) {
69              ByteBuffer byteBuffer = ByteBuffer.wrap(buffer);
70              int bytesWritten = 0;
71              while (bytesWritten < TOTAL_SIZE) {
72                  random.nextBytes(buffer);
73                  byteBuffer.clear();
74                  channel.write(byteBuffer);
75                  bytesWritten += BUFFER_SIZE;
76              }
77          }
78      }
79
80      private void readLargeFile(Path filePath) throws IOException {
81          byte[] buffer = new byte[BUFFER_SIZE];
82          try (SeekableByteChannel channel =
83              Files.newByteChannel(filePath, StandardOpenOption.READ)) {
84              ByteBuffer byteBuffer = ByteBuffer.wrap(buffer);
85              while (channel.read(byteBuffer) != -1) {
86                  byteBuffer.clear();
87              }
88          }
89      }
90  }
91  }
92
93  public interface ProductRepository extends JpaRepository<Product, Integer> {}
```

Listing 3. Klasa TestService i TestController aplikacji Web MVC

Załącznik 3. Zależności projektu Web MVC

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.4.0</version>
    <relativePath/>
  </parent>
  <groupId>pw.mgr</groupId>
  <artifactId>web-mvc-app</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
  <name>Web MVC</name>
  <developers>
    <developer>
      <name>Michal Tarka</name>
    </developer>
  </developers>
  <properties>
    <java.version>21</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

    <dependency>
```

```
43        <groupId>io.micrometer</groupId>
44        <artifactId>micrometer-registry-prometheus</artifactId>
45      </dependency>
46
47      <dependency>
48        <groupId>org.postgresql</groupId>
49        <artifactId>postgresql</artifactId>
50      </dependency>
51
52    </dependencies>
53
54    <build>
55      <plugins>
56        <plugin>
57          <groupId>org.springframework.boot</groupId>
58          <artifactId>spring-boot-maven-plugin</artifactId>
59        </plugin>
60      </plugins>
61    </build>
62
63  </project>
```

Listing 4. Zależności projektu Web MVC

Załącznik 4. Kod podejscia reaktywnego

```java
@RestController
public class TestController {

    private final TestService testService;

    public TestController(TestService testService) {
        this.testService = testService;
    }

    @GetMapping("/api/hello")
    public Mono<String> hello() {
        return Mono.just("Hello World!");
    }

    @GetMapping("/api/db")
    public Mono<String> getData() {
        return testService.queryDatabase()
                .thenReturn("OK");
    }

    @GetMapping("/api/file")
    public Mono<String> getFile() {
        return testService.performIOOperation(Paths.get("/tmp"))
                .thenReturn("OK");
    }

    @GetMapping("/api/delay")
    public Mono<String> getDelay(
            @RequestParam(name = "delay", required = false, defaultValue = "500")
            long delay) {
        return Mono.delay(java.time.Duration.ofMillis(delay))
                .then(Mono.just("OK"));
    }
}

@Service
public class TestService {

  private static final int BUFFER_SIZE = 4096;
  private static final int TOTAL_SIZE = 4096;

  private final ProductRepository productRepository;

  public TestService(ProductRepository productRepository) {
    this.productRepository = productRepository;
  }

  public Mono<Void> queryDatabase() {
    return productRepository.findAll().collectList().then();
  }

  public Mono<Void> performIOOperation(Path directory) {
    return Mono.fromCallable(() -> Files.createDirectories(directory))
        .then(generateFileName(directory))
        .flatMap(this::writeLargeFile)
        .flatMap(this::readLargeFile)
        .flatMap(this::deleteFile)
        .timeout(Duration.ofSeconds(30))
        .doOnError(e -> System.err.println(
            "[ERROR] Operation failed: " + e.getMessage()));
```

```
61  }
62
63  private Mono<Path> generateFileName(Path directory) {
64    return Mono.fromCallable(() -> directory.resolve(
65        "testfile_" + UUID.randomUUID() + ".dat"));
66  }
67
68  private Mono<Path> writeLargeFile(Path filePath) {
69    return Mono.using(
70        () -> AsynchronousFileChannel.open(filePath, StandardOpenOption.CREATE,
71            StandardOpenOption.WRITE),
72        fileChannel -> {
73          Random random = new Random();
74          Flux<ByteBuffer> buffers = Flux.range(0, TOTAL_SIZE / BUFFER_SIZE)
75              .map(i -> {
76                byte[] bytes = new byte[BUFFER_SIZE];
77                random.nextBytes(bytes);
78                return ByteBuffer.wrap(bytes);
79              });
80
81          return buffers
82              .concatMap(buffer -> writeToFile(fileChannel, buffer))
83              .then(Mono.just(filePath));
84        },
85        this::closeChannel
86    );
87  }
88
89  private Mono<Void> writeToFile(AsynchronousFileChannel fileChannel,
90                                 ByteBuffer buffer) {
91    return Mono.create(sink -> fileChannel.write(buffer, buffer.position(),
92        buffer, new CompletionHandler<Integer, ByteBuffer>() {
93          @Override
94          public void completed(Integer result, ByteBuffer attachment) {
95            sink.success();
96          }
97
98          @Override
99          public void failed(Throwable exc, ByteBuffer attachment) {
100           sink.error(exc);
101         }
102       }));
103 }
104
105 private Mono<Path> readLargeFile(Path filePath) {
106   return Mono.using(
107       () -> AsynchronousFileChannel.open(filePath, StandardOpenOption.READ),
108       fileChannel -> readFromFile(fileChannel)
109           .then(Mono.just(filePath)),
110       this::closeChannel
111   );
112 }
113
114 private Mono<Void> readFromFile(AsynchronousFileChannel fileChannel) {
115   return Mono.create(sink -> {
116     ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
117     readChunk(fileChannel, buffer, 0, sink);
118   });
119 }
120
121 private void readChunk(AsynchronousFileChannel fileChannel, ByteBuffer buffer,
122                        long position, MonoSink<Void> sink) {
```

```
123     fileChannel.read(buffer, position, buffer, new CompletionHandler<>() {
124       @Override
125       public void completed(Integer bytesRead, ByteBuffer attachment) {
126         if (bytesRead == -1) {
127           sink.success();
128         } else {
129           buffer.clear();
130           readChunk(fileChannel, buffer, position + bytesRead, sink);
131         }
132       }
133
134       @Override
135       public void failed(Throwable exc, ByteBuffer attachment) {
136         sink.error(exc);
137       }
138     });
139   }
140
141   private Mono<Void> deleteFile(Path filePath) {
142     return Mono.fromRunnable(() -> {
143       try {
144         Files.deleteIfExists(filePath);
145       } catch (IOException e) {
146         throw new RuntimeException(e);
147       }
148     }).subscribeOn(Schedulers.boundedElastic()).then();
149   }
150
151   private void closeChannel(AsynchronousFileChannel channel) {
152     try {
153       channel.close();
154     } catch (IOException e) {
155       System.err.println("[ERROR] Failed to close channel: " + e.getMessage());
156     }
157   }
158 }
159
160 public interface ProductRepository extends ReactiveCrudRepository<Product, Integer> {}
```

Listing 5. Klasa TestService i TestController aplikacji Webflux

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.4.0</version>
    <relativePath/>
  </parent>
  <groupId>pw.mgr</groupId>
  <artifactId>webflux-app</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
  <name>WebFlux</name>
  <developers>
    <developer>
      <name>Michal Tarka</name>
    </developer>
  </developers>
  <properties>
    <java.version>21</java.version>
  </properties>
  <dependencies>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-r2dbc</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

```

```
43      <dependency>
44        <groupId>io.micrometer</groupId>
45        <artifactId>micrometer-registry-prometheus</artifactId>
46      </dependency>
47
48      <dependency>
49        <groupId>org.postgresql</groupId>
50        <artifactId>r2dbc-postgresql</artifactId>
51      </dependency>
52
53    </dependencies>
54
55    <build>
56      <plugins>
57        <plugin>
58          <groupId>org.springframework.boot</groupId>
59          <artifactId>spring-boot-maven-plugin</artifactId>
60        </plugin>
61      </plugins>
62    </build>
63
64  </project>
```

Listing 6. Zależności projektu Webflux